IN-61-CR
217901
11 P

Benchmarking and Performance Analysis of the CM-2

David W. Myers
George B. Adams III

December, 1988

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.19

NASA Cooperative Agreement Number NCC 2-387

(NASA-CR-185406) BENCHMARKING AND PERFORMANCE ANALYSIS OF THE CM-2 (Research Inst. for Advanced Computer Science) 114 p CSCL 09B

N89-26419

Unclas G3/61 0217901

Research Institute for Advanced Computer Science

Benchmarking and Performance Analysis of the CM-2

*David W. Myers**
*George B. Adams II**

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.19
December, 1988

A suite of benchmarking routines testing communication, basic arithmetic operations, and selected kernel algorithms written in *LISP and PARIS was developed for the CM-2. Experiment runs are automated via a software framework that sequences individual tests, allowing for unattended overnight operation. Multiple measurements are made and treated statistically to generate well-characterized results from the noisy values given by **cm:time**. The results obtained provide a comparison with similar, but less extensive, testing done on a CM-1. Tests have been chosen to aid the algorithmist in constructing fast, efficient, and correct code on the CM-2, as well as gain insight into what performance criteria are needed when evaluating parallel processing machines.

The work reported herein was supported in part by Cooperative Agreement NCC2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA) and the Defense Advanced Research Projects Agency (DARPA). The contents of this document do not represent the official position of NASA or DARPA.

*The authors are with the School of Electrical Engineering, Purdue University, West Lafayette, Indiana. Work performed while visitors at RIACS and performed at the School of Electrical Engineering, Purdue University.

TABLE OF CONTENTS

1 Introduction

The impetus for undertaking an analysis of the CM-2 comes from several sources. First, many of the researchers at RIACS need timing information about various aspects of the CM-2. Much of the desired information is not readily available. Algorithmists can use this information to make prudent choices when writing code targeted for the CM-2. Secondly, the measurements taken on the CM-2 help to characterize aspects of the performance of the machine. Lastly, performance measures are needed to fully understand the consequences of design decisions made during development of the machine. Hence, this knowledge will aid the design process of future parallel processing machines.

The CM-2 is a single instruction stream, multiple data stream (SIMD) computer. The CM-2 system is comprised of three functional subsystems (see Figure 1): the control unit , which includes a front-end host computer and an interface to the "computational engine" of the CM-2 system called the sequencer; the processor/memory pairs (PEs); and the interconnection network. The control unit provides the user interface, executes serial portions of the users' CM-2 programs, and supplies the flow of instructions and data into the CM-2 for execution of the parallel portions of code. Currently, supported host processors include the Symbolics 3600 and the DEC VAX 8350.

CM-2 programming languages include *Lisp (pronounced *star lisp*) and C* (pronounced *see star*), which provide extensions to Common Lisp and to C, respectively, to support the writing of parallel code. These two high-level languages can include calls to the CM-2 assembly language Paris (**PAR**allel **I**nstruction **S**et).

Each PE consists of a 3-input bit serial ALU. Hence, the simplicity of the design of each PE permits sixteen PEs to be located on one chip. Each chip is a physical node in a twelve dimensional hypercube interconnection network. Sixty-four kilobits of bit-addressable memory are allocated to each PE, which means that a full CM-2 (64K PEs) contains 512 megabytes of memory.

Most of the experiments in this work do not pertain to the control unit directly. Instead, they focus on the parallel portion of the CM-2 system: the PEs and the interconnection network.

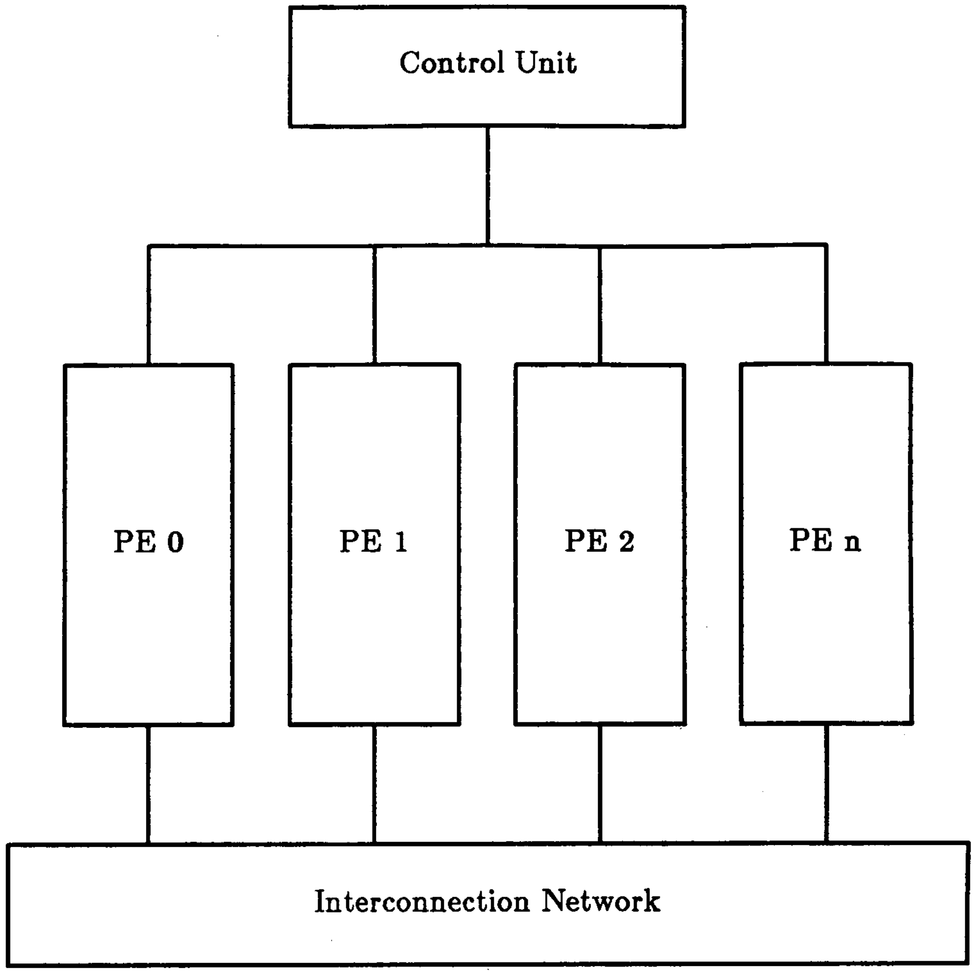


Figure 1: Typical SIMD computer structure.

2 Methodology

The approach taken in this work is to conduct the analysis in the manner used in the traditional sciences such as physics or as chemistry. All measurements reported are reproducible results. Experiments conducted to test a hypothesis include a carefully constructed control. All raw measurement data, statistical reduction routines, and experiment codes are available to the interested reader by contacting the corresponding author, Adams.

2.1 Experiment Design

Timing data can be used to help compare the CM-2 with other machines. However, care must be taken if a meaningful comparison is to be made. This is so in part because it is not fully understood what parameters should be used when comparing two parallel processing computers, or when comparing a parallel computer to a serial "conventional" computer.

When deciding what experiments to run on the CM-2, several factors are considered. First, in many cases tests are designed explicitly to exercise a specific portion of the CM-2 or a particular software function. Secondly, in other cases a detail of the machine was not described in the documentation or in the literature [HIL85, TM86, TM87a, TM87b], a hypothesis about the aspect was made, and then an experiment was designed to test this hypothesis. For example, the existence of simultaneous, bidirectional communication over the hypercube interconnection network between adjacent processors is not explicitly listed. A hypothesis was formulated, an experiment was designed, a test was run, and simultaneous, bidirectional communication capability was verified. Finally, specific algorithms were employed to assess the performance of the CM-2. For instance, a digital image smoothing algorithm tests several important properties of the CM-2. These include nearest neighbor communications, integer addition, and integer division.

2.2 Code

All measurements were written using the *Lisp programming language. The magnitude of the benchmarking and of the performance analysis task undertaken demanded automation of as much of the work as feasible and required good programming practice to contain to a single module those aspects of the codes that changed between measurements or experiments. The test codes consist of four modules: the main body, the timing routines, the message concentration function, and the statistics function.

2.2.1 Main Body

The main body of each test suite carries out two tasks. These tasks include opening/closing the output files and calling the timing routines, which embody the measurements and experiments to be conducted.

Each test file manipulates two data files. The first data file contains all the timing measurements, the "*raw*" data, used by the statistics function. The second data file contains the statistics reported about each timing routine. Data are written into this file from the statistics function and the message concentration function of each test file (see Sections 2.2.3 and 2.2.4, respectively). As mentioned above, the second primary task managed by the main body involves calling the timing routines. Two parameters are passed to each timing routine. The first parameter, *time-loop*, dictates the number of sample times to be recorded for statistical treatment (100 was used for each experiment); the second parameter, *time-test*, determines the number of executions of an instruction during any one *time-loop* timing run.

2.2.2 Timing Routines

Each timing routine embodies one measurement. An experiment thus consists of at least two timing routines (measurements), one to act as the control and one to test the hypothesis. To date, the types of tests run include communication, arithmetic, and selected kernel algorithms (see Section 3 for further details about specific tests). All timing routines share a common skeleton (see Figure 2), and differ only in the details pertaining directly to the test. A description of the test to be executed is written to the raw data file and to the statistics file, and a vector is created to store the *time-loop* execution times. Notice the two **do** loops in Figure 2. The outer **do** loop corresponds to the *time-loop* samples; whereas, the inner **do** loop is needed to compensate for certain characteristics of the timing routine used (**cm:time**, see Section 2.2.4). Finally, after all timing runs executed, the message concentration function and the statistics function are called.

```
;;;
;;; Timing #example
;;; Date
;;; Written by:              David Myers
;;; Description:  This is the skeleton used for the timing tests.
;;; Active Processors:
;;; Size of Data Used:
;;;
(*defun time-?? (time-loop test-loop)
 (format *fp* "~%SHORT DESCRIPTION~%")
 (format *data* "~%SHORT DESCRIPTION~%")
 (*all
  (let ((values 0))
   (setq values (make-array '(100) :fill-pointer 0))

   ;;
   ;; Test set-up code goes here
   ;;

   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)

      ;;
      ;; Code to be timed goes here
      ;;

     :return-statistics-only-p t)
    (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop)))))
```

Figure 2: Timing skeleton.

2.2.3 Message Concentration Function

The concentration function used here is the same function used by Bill O'Farrell in his tests at Syracuse University on the CM-1 [OFA87]. This function was used in the CM-2 study primarily during experiment debugging as a tool to help verify that tests are functioning as intended. The information reported by the concentration function includes:

1. The number of active PEs.

2. The number of PEs receiving messages.

3. The average number of messages received. This number is equal to $\frac{\#\text{PEs receiving msgs}}{\#\text{active PEs}}$.

4. The maximum of the number of messages received by each active PE and the number of PE(s) receiving this many messages.

The information from points 1 through 4 above and the information from the statistics function are written into a log file along with a descriptive title of the test. For each run, an additional data file is created to store the data returned by **cm:time**.

2.2.4 Statistics Function

Statistics on the numbers reported from each timing routine are computed by the statistics function (see Appendix A). The parameters needed by the statistics function include the number of sample values, *time-loop*, and the sample values, *values*. The statistics calculated entail the maximum, minimum, average, and standard deviation. The maximum and the minimum of the samples are computed using the Lisp functions **max** and **min**, respectively. The average is computed as follows:

$$\text{average} = \sum_{i=1}^{time-loop} \frac{values_i}{time-loop} \quad . \tag{1}$$

The standard deviation is computed using:

$$\text{std-dev} = \sum_{i=1}^{time-loop} \left(\frac{x_i^2}{n-1} \right) - \left(\frac{n}{n-1} \right) \bar{x}^2 \quad . \tag{2}$$

The function supplied by Thinking Machines Corp. for timing executions on the CM-2 is **cm:time**. The first time **cm:time** is called during a user session, the ratio of CM-2 cycles to host machine cycles is determined. Then, after all the CM-2 internal buffers have been cleared of previous macroinstructions, the system time is recorded and the CM-2 idle timer is reset (note: whenever the CM waits for instructions from the front end, the idle timer counter is incremented). Next, **cm:time** runs the *Lisp form it was given, and on completion of the form, the idle time and the system time are once again recorded. From this information, the CM-2 active time, the front end time, and the utilization are estimated.

A primary concern when conducting measurements is the accuracy and precision of results being reported. Early experimentation determined that **cm:time** is not precise for short program runs, because it depends on values obtained from the front-end system clock. The system call used to access the clock runs with non-deterministic timing due to other processes on the front end.

Because of the overhead involved in activating **cm:time**, this utility is not accurate when timing only several (Paris) instructions. Thinking Machines does include a constant factor for the overhead time in the code for **cm:time**; however, it does not alleviate timing inaccuracy. To overcome the limited precision of the results reported by **cm:time**, each code segment is executed repetitively within **cm:time**. Testing determined that the numbers reported are quite precise if the total elapsed time of the form being timed is on the order of several seconds. Therefore, the variable *cm-time* is bound to the return value corresponding to the CM-2 active time, and the result stored in the vector *values* represents the instruction/task execution rate of

$$\text{rate} = \frac{cm-time}{test-loop} \quad . \tag{3}$$

The statistics calculated are used to verify the correctness of the data gathered. The maximum and the minimum show the range of times collected; with the minimum giving the nearest to optimal execution time (i.e., each instruction should always execute in the same number of cpu cycles). The average time indicates the expected execution time, and when the average time is considered with the standard deviation, the accuracy of the timings can be readily deduced. See Section 3.1 for tabulated data values.

2.3 Experiment Execution

The measurements and the experiments have been run with a DEC VAX 8350 as the control unit (front end). Because the control unit is not a single-user machine, the loading on the VAX can have a direct impact on the performance of programs executing on the CM-2. Therefore, to reduce this possibility and to avoid user contention of the CM-2 during normal working hours, all experiments were run at night.

To facilitate running the experiments at night, the Unix **at** command was initially employed unsuccessfully. This lead to the development of the shellscripts shown in Appendix C. There are two versions of these shellscripts: one for V4.3 software (**old-night**), and one for V5.0 software (**night**).

The shellscript **night** (**old-night**) is similar to the Unix **at** command. It accepts as command-line arguments the desired time (24 hour time) of execution, and compares the entered time to the actual time every 15 minutes. When the start time is reached, the shellscript **CMrun** (**old-CMrun**) loads the file (**foo**) containing the desired LISP forms to be executed and pipes it into **starlisp** (**old-starlisp**). For example, to execute the program specified in file **foo** at midnight:

% night 00 00& .

At midnight or shortly thereafter, the program will be executed with V5.0 software.

3 Results and Analysis

3.1 Tables

Shown on the following four pages are tables of the initial results obtained. The first column in each table corresponds to the test number of each timing experiment. These numbers can be used to locate specific routines in Appendix C (e.g., **time-testXX** corresponds to test number XX in **final-syracuse.lisp**, and **time-XX** corresponds to test number XX in one of the other files (each test has a distinct number)). In addition, a more detailed description of the tests is included with each timing routine as part of the comment header. Also, additional description and analysis of the experiments completed at RIACS are located in Section 3.3 and in Section 3.4. Table 1 reports the results of experiments done at RIACS, and Table 2 reports the results from experiments run on the CM-1 at Syracuse and the results from the same experiments run on the CM-2 at RIACS.

Table 1: CM-2 timing experiments.

CM-2 Timing Experiments - Version 4.3 Software							
Tests run with a VAX 8350 front end - 8192 PEs used							
All measurements listed in units of milliseconds							
Test Number	Description	Instruction Used	Size of Data	Average Time	Standard Deviation	Max. Time	Min. Time
1	swap	cm:send	32	.1686	.0021	.1729	.1572
2	1 way send	cm:send	32	.1668	.0016	.1714	.1624
3	max dist.	cm:send	32	.1689	.0026	.1755	.1576
4	inner product	*Lisp	32	2.030	.018	2.068	1.945
5	HD = 13	*pset	1	1.077	.008	1.081	1.012
5.1	HD = 4	*pset	1	.3312	.0003	.3321	.3297
6	HD = 13	*pset	2	1.094	.004	1.097	1.068
6.1	HD = 4	*pset	2	.3368	.0023	.3375	.3145
7	HD = 13	*pset	4	1.120	.001	1.122	1.114
7.1	HD = 4	*pset	4	.3451	.0006	.3458	.3432
8	HD = 13	*pset	8	1.171	.007	1.174	1.104
8.1	HD = 4	*pset	8	.3630	.0005	.3638	.3607
9	HD = 13	*pset	16	1.280	.003	1.282	1.256
9.1	HD = 4	*pset	16	.3973	.0006	.3980	.3942
10	HD = 13	*pset	32	1.160	.007	1.163	1.090
10.1	HD = 4	*pset	32	.3660	.0005	.3667	.3646
11	HD = 13	*pset	64	1.823	.002	1.825	1.815
11.1	HD = 4	*pset	64	.5448	.0024	.5457	.5218
12.1	HD = 1	*pset	80	.9372	.0005	.938	.9361
12.2	HD = 2	*pset	80	.937	.0024	.9381	.9146
12.3	HD = 3	*pset	80	.9371	.0024	.9381	.9142
12.4	HD = 4	*pset	80	.9372	.001	.938	.9354
12.5	HD = 5	*pset	80	3.083	.0017	3.086	3.076
12.6	HD = 6	*pset	80	3.083	.0017	3.085	3.079
12.7	HD = 7	*pset	80	3.081	.015	3.085	2.936
12.8	HD = 8	*pset	80	3.083	.0019	3.085	3.075
12.9	HD = 9	*pset	80	3.083	.0025	3.086	3.054
12.10	HD = 10	*pset	80	3.082	.0067	3.085	3.015
12.11	HD = 11	*pset	80	3.083	.004	3.086	3.040
12.12	HD = 12	*pset	80	3.083	.004	3.086	3.076
12.13	HD = 13	*pset	80	3.083	0.0	3.086	3.078
13	HD = 13	*pset	128	4.678	.007	4.683	4.615
13.1	HD = 4	*pset	128	1.411	.0023	1.412	1.389
14	HD = 13	*pset	256	8.298	.0118	8.302	8.220
14.1	HD = 4	*pset	256	2.495	0.0	2.500	2.491
15	calculation	*Lisp	float	5.922	.528	61.140	58.023
19	random	random!!	13	1.673	.011	1.699	1.630
20	shffl. exch.	*Lisp	32	1.811	.003	1.814	1.787
21	shffl. exch.	*Lisp	32	6.111	.039	6.189	5.808
22	s.e., bit shft	*Lisp	32	2.401	.0166	2.427	2.306
23	s.e., calc. incl.	*Lisp	32	1.994	.041	2.096	1.870
24	shffl. exch.	*Lisp	32	1.919	.041	2.024	1.820
25	NEWS, 4	*Lisp	32	.594	.0035	.5985	.5610
26	NEWS, 8	*Lisp	32	1.679	.009	1.685	1.672

Table 1 (cont.): CM-2 timing experiments.

CM-2 Timing Experiments - Version 4.3 Software - Continued							
Tests run with a VAX 8350 front end - 8192 PEs used							
All measurements listed in units of milliseconds							
Test Number	Description	Instruction PEs	Size of Data	Average Time	Standard Deviation	Max. Time	Min. Time
27	Image smoothing	*Lisp	float	7.184	.026	7.202	7.165
28	PM2I, I = 0	*Lisp	32	.3652	0.0	.3652	.3651
28.1	PM2I, I = 1	*Lisp	32	.4308	.0003	.4310	.4306
28.2	PM2I, I = 2	*Lisp	32	.6342	.0011	.6350	.6333
28.3	PM2I, I = 3	*Lisp	32	.8308	.0003	.8312	.8305
28.4	PM2I, I = 4	*Lisp	32	1.356	0.0	1.356	1.356
28.5	PM2I, I = 5	*Lisp	32	1.356	0.0	1.356	1.356
28.6	PM2I, I = 6	*Lisp	32	1.287	0.0	1.287	1.287
28.7	PM2I, I = 7	*Lisp	32	1.223	0.0	1.223	1.223
28.8	PM2I, I = 8	*Lisp	32	1.223	0.0	1.223	1.223
28.9	PM2I, I = 9	*Lisp	32	1.224	.0005	1.224	1.223
28.10	PM2I, I = 10	*Lisp	32	1.226	.0008	1.226	1.225
28.11	PM2I, I = 11	*Lisp	32	1.159	0.0	1.59	1.159
28.12	PM2I, I = 12	*Lisp	32	1.159	.0007	1.159	1.158
29	NEWS, 8	*Lisp	32	13.926	.017	13.938	13.913
30	ring	*Lisp	32	.3659	.0006	.3663	.3655
31	ring, max inactive	*Lisp	32	.3653	.0002	.3654	.3652
32	unsigned +	PARIS	32	.0817	.0009	.0848	.0805
33	unsigned x	PARIS	32	.9802	.0058	.9835	.9238
34	unsigned -	PARIS	32	.0892	.0008	.0921	.0869
35	unsigned ÷	PARIS	32	1.212	.020	1.220	1.017
36	fp +	PARIS	float	.556	.0007	.5575	.5542
37	fp -	PARIS	float	.5578	.0036	.5596	.5242
38	fp x	PARIS	float	.534	.0036	.5358	.4988
39	fp ÷	PARIS	float	1.228	.0015	1.230	1.224
40.1	rev., HD = 1	*pset	80	3.079	.004	3.083	3.070
40.2	rev., HD = 2	*pset	80	3.078	.005	3.086	3.051
40.3	rev., HD = 3	*pset	80	3.078	.004	3.083	3.058
40.4	rev., HD = 4	*pset	80	3.077	.018	3.086	2.897
40.5	rev., HD = 5	*pset	80	3.078	.005	3.088	3.070
40.6	rev., HD = 6	*pset	80	3.078	.004	3.087	3.059
40.7	rev., HD = 7	*pset	80	3.078	.004	3.083	3.065
40.8	rev., HD = 8	*pset	80	3.077	.019	3.084	2.894
40.9	rev., HD = 9	*pset	80	3.079	.003	3.084	3.070
40.10	rev., HD = 10	*pset	80	3.078	.003	3.084	3.070
40.11	rev., HD = 11	*pset	80	3.074	.023	3.083	2.894
40.12	rev., HD = 12	*pset	80	3.075	.022	3.088	2.909
40.13	rev., HD = 13	*pset	80	3.074	.024	3.087	2.878
42.1	1 PE	*pset	80	.388	.002	.392	.378
42.2	2 PE	*pset	80	.565	.003	.570	.652
42.3	3 PE	*pset	80	.739	.015	.745	.603
42.4	4 PE	*pset	80	.914	.014	.919	.781
42.5	5 PE	*pset	80	1.082	.014	1.087	.948
42.6	6 PE	*pset	80	1.305	.007	1.313	1.249
42.7	7 PE	*pset	80	1.484	.016	1.493	1.373
42.8	8 PE	*pset	80	1.664	.011	1.673	1.592
42.9	9 PE	*pset	80	1.833	.022	1.843	1.641
42.10	10 PE	*pset	80	2.012	.007	2.027	1.418
42.11	11 PE	*pset	80	2.197	.011	2.204	2.105
42.12	12 PE	*pset	80	2.376	.029	2.386	2.108
42.13	13 PE	*pset	80	2.546	.031	2.559	2.280
42.14	14 PE	*pset	80	2.725	.028	2.736	2.453
42.15	15 PE	*pset	80	2.911	.006	2.911	2.898
42.16	16 PE	*pset	80	3.088	.017	3.099	2.972

Table 2: Timing comparison of CM-1 [OFA87] versus CM-2

Timing Comparison CM-1 vs CM-2 All measurements listed in units of milliseconds Test Number	CM-1 execution time [OFA87]	CM-2 execution time	CM-2 std dev	CM-2 max	CM-2 min
1	.306	.289	.0042	.2901	.2581
2	NOT *	D	O	N	E
3	2.9	2.85	.0286	2.86	2.57
4	2.6	1.34	.0131	1.35	1.22
5	2.4-2.5	3.91	.0053	3.915	3.858
6	2.4-2.9	1.26	.0056	1.264	1.207
7	1.9-2.0	1.44	.0003	1.446	1.438
8	1.85-1.92	1.28	.0012	1.281	1.276
9	NOT	D	O	N	E
10	NOT	D	O	N	E
11	2.5-2.6	1.18	.0019	1.183	1.169
12	1.6-1.7	1.11	.0147	1.121	.977
13	NOT	D	O	N	E
14	2.4-2.6	1.18	.0002	1.183	1.179
15	1.3	.4369	.0056	.4397	.3818
16	.420	.4376	.0023	.4383	.4146
17	2.5-2.8	1.34	.0024	1.349	1.334
18	1.3	1.43	.0015	1.433	1.426
19	1.3	1.43	.0015	1.432	1.331
20	1.7	1.43	.0012	1.433	1.425
21	.869	.930	.0059	.9333	.8727
22	1.7	1.43	.0014	1.432	1.419
23	1.7	1.43	.0016	1.431	1.424
24	1.7	1.43	.0095	1.432	1.335
25	1.7	1.43	.0016	1.432	1.423
26	2.9	1.76	.0024	1.764	1.750
27	5.4	1.76	.0037	1.766	1.743
28	10.4	2.08	.0083	2.105	2.057
29	20.1	2.24	.0178	2.273	2.111
30	2.1	1.19	.0155	1.199	1.039
31	1.06	1.12	.0008	1.199	1.193
32	.646	.608	.0004	.6095	.6066
33	.192	.441	.0012	.4416	.4297
34	.268	.441	0.0	.4419	.4407
35	.469-.589	.444	.0005	.4442	.4424
36	NOT	D	O	N	E
37	NOT	D	O	N	E
38	NOT	D	O	N	E
39	.350-.409	.3734	.0005	.3744	.3722
40	NOT	D	O	N	E
41	NOT	D	O	N	E
42	NOT	D	O	N	E
43	NOT	D	O	N	E
44	NOT	D	O	N	E
45	NOT	D	O	N	E
46	NOT	D	O	N	E
47	8.9	6.64	.070	6.745	6.080
48	2.4	2.70	.001	2.702	2.687
49	2.8	2.85	.004	2.858	2.837
50	3.8	3.70	.007	3.720	3.680

* Tests marked NOT D O N E are those that, while meaningful for the CM-1, are irrelevant or not possible given the different hardware and software of the CM-2.

Table 2 (cont.): Timing comparison of CM-1 [OFA87] versus CM-2

n.sp 1

Timing Comparison CM1 vs CM2, Continued

All measurements listed in units of milliseconds

Test Number	CM-1 execution time	CM-2 execution time	CM-2 std dev	CM-2 max	CM-2 min
51	5.9	5.41	.012	5.436	5.388
52	17.8	15.5	1.58	16.05	13.78
53	2300	338	38.2	349.4	197.8
54	.723	.7014	.0038	.7040	.6647
55	.671	.6647	.0027	.6659	.6377
56	.525	.5077	.0005	.5088	.5061
57	.671	.6646	.0038	.6661	.6382
58	NOT *	D	O	N	E
59	NOT	D	O	N	E
60	NOT	D	O	N	E
61	NOT	D	O	N	E
62	NOT	D	O	N	E
63	.675	.5135	.0028	.515	.4863
64	NOT	D	O	N	E
65	NOT	D	O	N	E
66	NOT	D	O	N	E
67	NOT	D	O	N	E
68	NOT	D	O	N	E
69	NOT	D	O	N	E
70	NOT	D	O	N	E
71	.0381	.0430	.0002	.0434	.0425
72	.0381	.0430	0.0	.0432	.0428
73	.117	.1034	.0005	.1037	.0981
74	1.13	1.045	.0012	1.048	1.040
75	1.69	1.707	.0035	1.711	1.676
76	NOT	D	O	N	E

* Tests marked NOT D O N E are those that, while meaningful for the CM-1, are irrelevant or not possible given the different hardware and software of the CM-2.

3.2 Communication Tests Analysis

time-01 and **time-02**

time-01 and **time-02** are the first two experiments designed for the CM-2. The significance of these two experiments lies not in the knowledge gained about the CM-2, but in the insight they give to the methodology used when constructing the experiments.

Prior to conducting these two experiments, the bidirectional communication capabilities of the CM-2, if any, were unknown to us. To investigate this problem further, two experiments were used. The first experiment, **time-01**, reports the time for the CM-2 to complete bidirectional communication operation where the active PEs are those satisfying the following condition

$$\mathit{self\_address} \bmod 16 = 0 \quad . \tag{4}$$

This ensures that only one PE per chip is active. Each active PE calculates a destination address as follows

$$\mathit{dest\_address} = \mathit{self\_address} \oplus 16 \quad . \tag{5}$$

Hence, each active PE sends to a destination PE that is a Hamming distance of one away through the network. In addition, considering Equation (4) and Equation (5) above, it is evident that each active PE transmits a message to a destination PE satisfying the following equation

$$\mathit{source\_PE\_address} \bmod 32 = \mathit{dest\_PE\_address} \bmod 32 \quad . \tag{6}$$

The constraints above create a situation in which each active PE is communicating with a single unique PE in the hypercube (i.e., bidirectional communication). This is shown for a 3-dimensional hypercube in Figure 3 below.

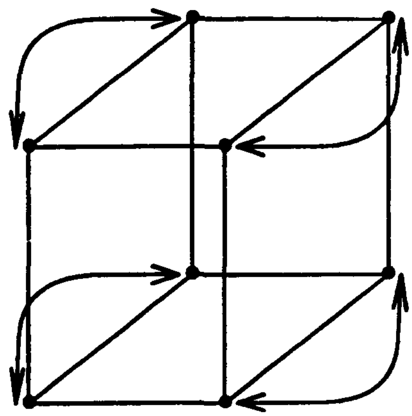


Figure 3: Communication pattern for 3-dimensional hypercube with experiment **time-01**.

The second test, **time-02**, is very similar to the first test except only half of the active PEs from the first test are used in the second test. The active set of PEs are those satisfying

$$self-address \bmod 32 = 0 \tag{7}$$

and destination addresses are calculated as shown by Equation (5). These constraints create a situation in the CM-2 in which one-way communication occurs through the network instead of two-way communication. This is shown for a 3-dimensional hypercube in Figure 4.

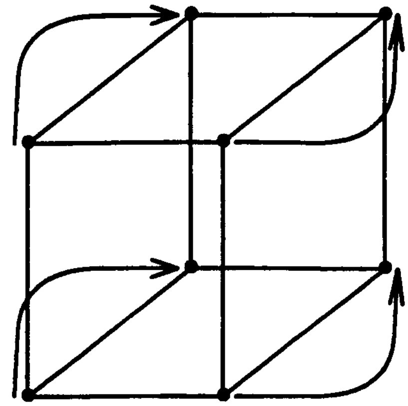


Figure 4: Communication pattern for 3-dimensional hypercube with experiment **time-02**.

The times reported by **test-01** and by **test-02** are equal; thus demonstrating that the CM-2 does have bidirectional communication capability.

time-03

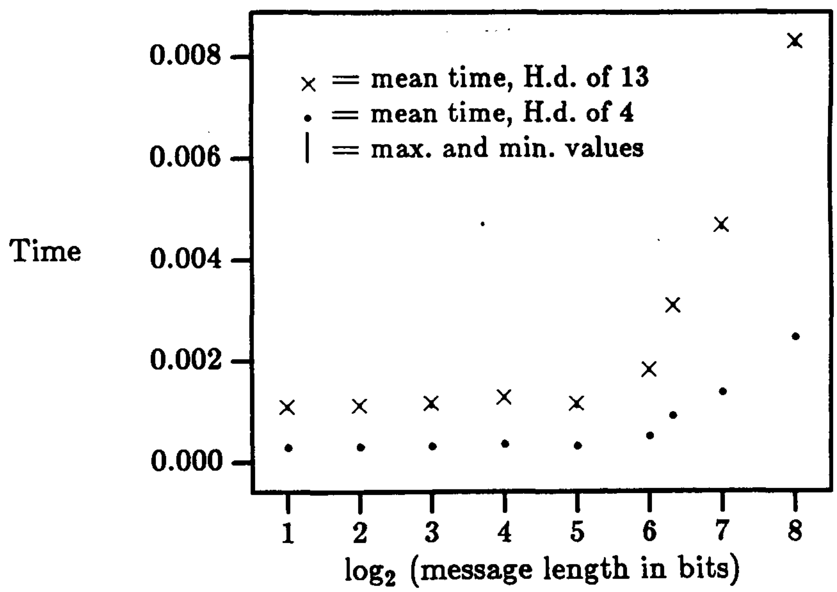
This test reports the amount of time required to send a 32 bit message from PE (0) to PE (*cm:*user-cube-address-limit** $-$ 1). The time reported by **time-03** is equal to the time reported by **time-01** and by **time-02**. This can be attributed to the lack of contention for resources in the network. All PEs are sending to a unique PE and the number of active PEs is small; hence, there is never more than one message needing to be sent from any single router at any given instant.

time-05 through **time-14**

These experiments test communications for two different Hamming distances (4 and 13) over a wide range of message sizes (see Graph 1). The message sizes are all powers of two except for the 80 bit message size that corresponds to the number of bits in an IEEE Floating Point Standard double extended precision number[IEE85]. A Hamming distance of four is the largest Hamming distance possible while maintaining on-chip communications; whereas, a Hamming distance of thirteen is the maximum Hamming distance within a hypercube of 8K nodes.

Graph 1: Hamming distance communication experiment,

tests **time-05** through **time-14**.



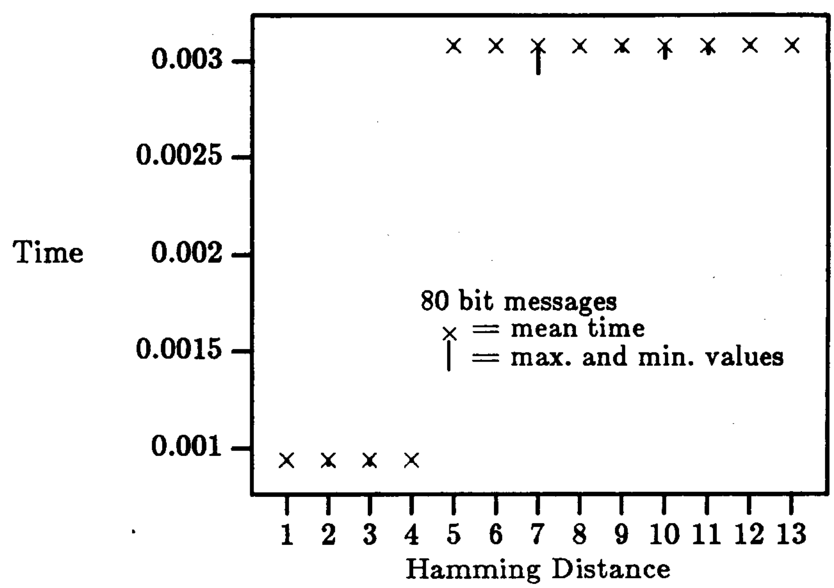
The size of the message being sent has an affect on the time required to complete a message transfer. This is to be expected. However, the time required to transmit messages increases with the message size up to a message size of 29; at this point the time to transmit a message decreases, and then starts to rise again. In addition, Graph 1 shows that on-chip communication is much faster than network communication.

time-12.1 through **time-12.13**

This group of experiments shows the time required for 80-bit messages being sent over varying Hamming distances (see Graph 2).

Graph 2: Hamming distance communication experiment,

tests **time12.1** through **time-12.13**.



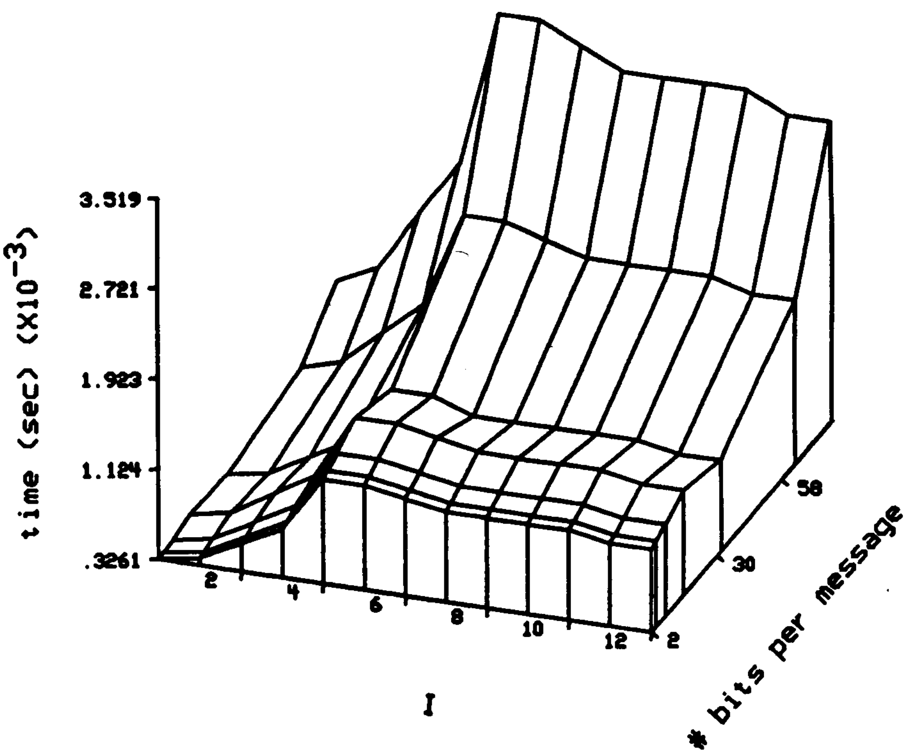
To calculate a destination address that is a Hamming distance of Y from a source address X, the lower Y bits of X are inverted. This explains the two levels shown Graph 2. The method for calculating destination addresses mentioned above yields on-chip communications for Hamming distances between 1 and 4; whereas, off-chip communications arise for Hamming distances between 5 and 13. Again this shows the speed advantage of on-chip communications versus off-chip communications.

time-28.1 through **time-28.12**

This experiment tests the ability of the CM-2 hypercube network to simulate the PM2I network [SIE85]. The results are shown in Graph 3 and in Graph 4.

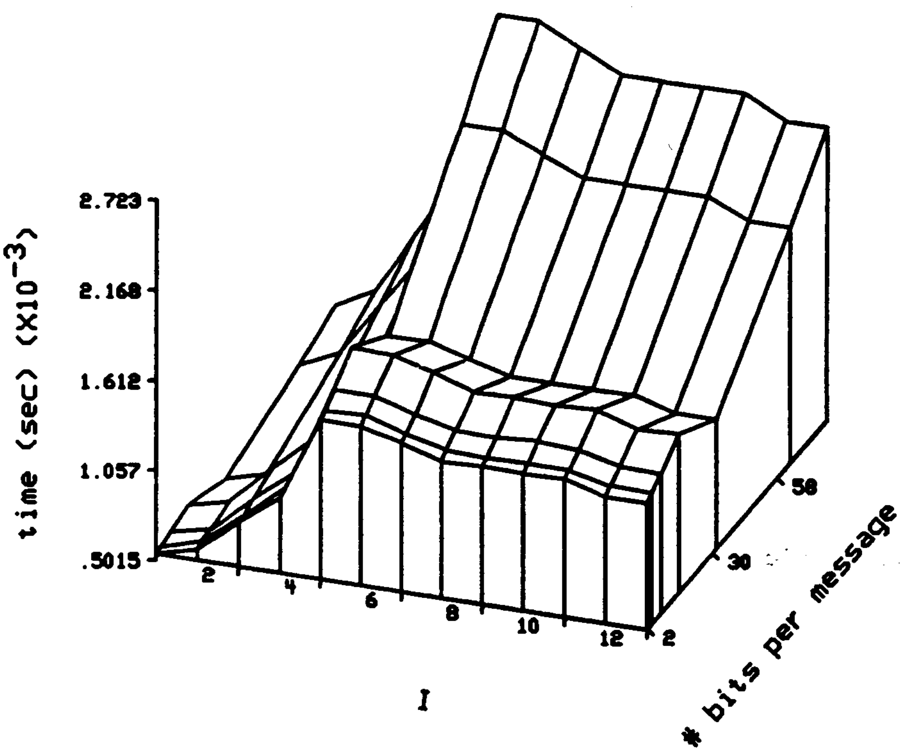
Graph 3: PM2I simulation (V4.3 software),

tests **time-28.1** through **time-28.12**.



Graph 4: PM2I simulation (V5.0 software),

tests **time-28.1** through **time-28.12**.



For small values of I, source PEs are sending to destination PEs which are are separated by a small absolute distance (i.e., 1, 2, 4 away). A smaller absolute distance implies more overall on-chip communications; hence, a smaller total communication time. This explains the shape of Graph 3 and of Graph 4 for small values of I.

The shape of the graph for the remaining values of I can be attributed to several factors. First, for greater values of I, more off-chip communication will occur. Secondly, the Hamming distance between the source PE and the destination PE affects the communication time. Recall that the PM2I network adds $\pm 2^I$ to every source PE address to calculate every destination PE address. Adding 2^I to a binary number entails adding 1 at position I + 1. If the bit in location I + 1 is a zero, a carry into bit position I + 2 does not occur, and the Hamming distance between the source/destination PEs is one. For example, adding one to eight yields:

$$\begin{array}{r} 01000 \\ +00001 \\ \hline 01001 \end{array}$$

However, if a one is present in bit position I + 1, a carry occurs into bit position I + 2. Depending on the number of ones in the higher bit positions, this carry-propagate could continue out to the most significant bit (MSB). For example, adding one to seven yields:

$$\begin{array}{r} 00111 \\ +00001 \\ \hline 01000 \end{array}$$

From the example above, it is apparent that each carry-propagate increases the Hamming distance between source/destination PEs by one. Consequently, the maximum on the two graphs arises when the carry-propagate affects the most source/destination pairs with the greatest Hamming distance separation.

It is obvious that the maximum Hamming distance between any one source/destination pair occurs when I = 4 because the greatest number of bits can be changed by a ripple carry from this position. However, this does not explain why the graphs drop off as I increases.

The average ripple carry distance when adding 1 to an n bit number is bounded above by $\log_2(n)$ [HWA79]. Therefore, an inverse relationship exists between I and n, and the largest average ripple carry distance occurs for I =

1. However, the ripple carry for small values of I affect only the lower order bits. This implies that most of the communications are occurring between PEs located on chip. For I greater than or equal to 4, the bits effected by the ripple carry are those corresponding to paths between chips. This explains the gradual increase in communication time up to I = 4, and the drop off in the graph for values of I greater than 4 can be attributed to a decrease in the average ripple carry distance (i.e., $\log_2(n)$ decrease as n decreases).

When comparing the PM2I simulation between the V5.0 software and the V4.3 software, it is immediately obvious that both graphs have the same shape. However, the communication times between corresponding operations are not equal. The V4.3 software transmits smaller messages faster than the V5.0 software, and the V4.3 software transmits larger messages much slower than the V5.0 software. This demonstrates that the V5.0 software transmits messages with more consistent performance than the V4.3 software.

time-30 and **time-31**

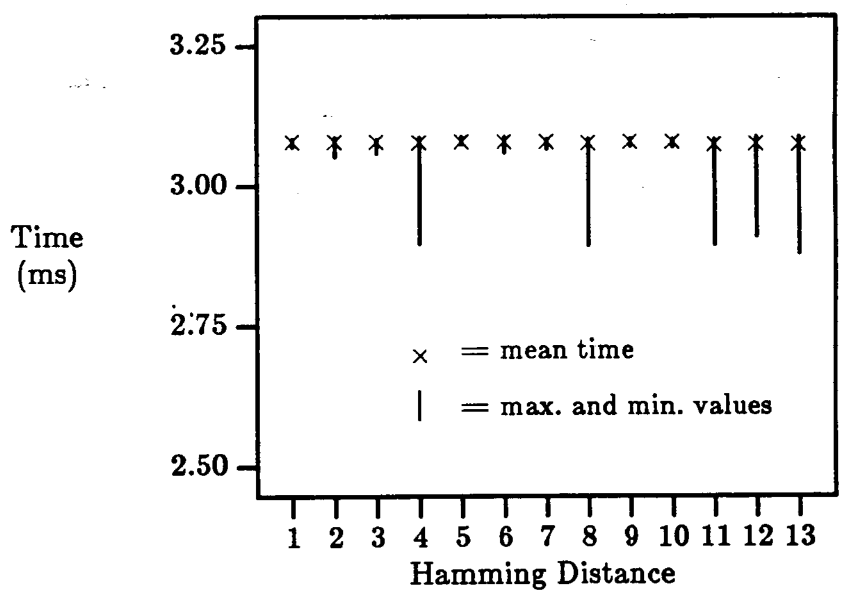
For these two tests, all the PEs are logically connected in a ring configuration, and each PE sends to the next PE in the ring. The PEs are numbered in the ring according to a binary reflected gray code [TFC86]. When this type of ordering is done in a hypercube network, neighboring PEs in the ring are separated by a Hamming distance of only one. Hence, only one dimension of the hypercube need be traversed before a message arrives at its destination. It is interesting to note that the time for ring communication is the same regardless if the ring is closed or if the ring is broken between PE 0 and the PE max-PE-address.

time-40

This is the companion test to **test-12**. In **test-12**, the source addresses are modified from the least significant bit to the most significant bit when calculating destination addresses; whereas, in **test-40**, the source addresses are modified from the msb to the lsb when calculating destination addresses. Graph 5 displays the results of this test.

Graph 5: Reverse Hamming distance test,

test **time-40**.



When comparing Graph 5 with Graph 2 it is apparent that the discontinuity in Graph 2 is not present in Graph 5. This is a direct result of the method used to calculate the destination addresses. Since, the destination addresses are calculated by applying the exclusive OR operation to the source address bits from the most significant bit to the least significant bit, all the communications are occurring between PEs located on different chips.

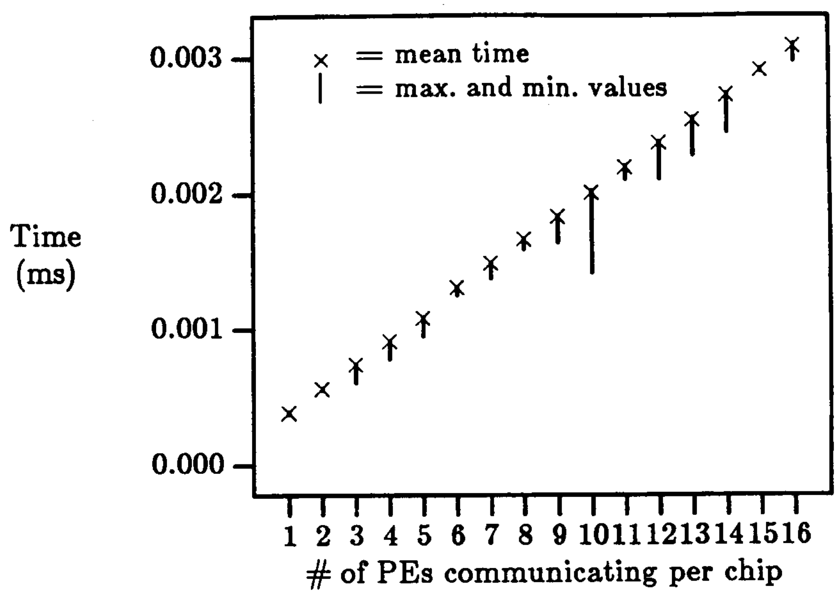
It is interesting to note that in both Graph 2 and in Graph 5, the communication time for different Hamming distances are equivalent. This is a result of the communication pattern occurring in the CM-2. In any given occurrence α of the Hamming distance test, each source/destination PE pair is separated by a Hamming distance of α. Hence, each message must traverse α dimensions of the hypercube. The communication pattern of this test is a bijection on the set of source PE addresses. This implies equal loading (number of messages) at each point during the communication steps. Also, the type of routing algorithm (i.e., non-deterministic) has an effect on the communication time.

time-42

This experiment tests the effect the number of active PEs per chip has on off-chip communication time (see Graph 6).

Graph 6: PE communicating off-chip,

test **time-42**.



As would be expected, the number of active PEs communicating off-chip does have a significant influence over the communication time. After analyzing the data it is apparent that the relationship between the number of PEs per chip involved in the communication and communication time is almost linear. In fact by using linear regression, this relationship can be approximated with the equation

$$\text{time} = 180.6 \times 10^{-6} \times (\#\ \text{active PEs/chip}) + 204.4 \times 10^{-6} \quad . \tag{8}$$

This relationship occurs because as the the number of PEs communicating off-chip increase, the time to service all the requests increase. Hence, the total communication time increases.

3.3 Algorithm Tests Analysis

time-04

The equation used for the inner product test is given below

$$\mathrm{IP} = \sum_{i=0}^{i=n-1} a_i b_i \quad . \tag{9}$$

Each PE_i contains both a_i and b_i. The product, $a_i b_i$, is computed by each PE, and then the sum-of-products is calculated using the *Lisp command **scan!!** with the result being stored in the PE with address *cm:*user-cube-address-limit* $-$ 1. The inner product is a good algorithm to choose for testing the CM-2 because it lends itself nicely parallelization on an SIMD machine. Multiplication, addition, and recursive doubling are all features of a parallel algorithm that are present in this test.

time-27

An image can be represented as a 2-dimensional array of pixels. Each pixel in the image is assigned a gray level and an (x,y) coordinate position within the image. To remove extraneous noise from the picture, image smoothing is performed. This is done by computing the average gray level for each non-border pixel and its eight nearest neighbors. Therefore, if each pixel in the image with coordinates i and j has a gray level associated with it, h(i,j), then the new gray level for the pixel at position (i,j) is

$$h_{new}(i,j) = \frac{1}{9} \sum_{i=-1}^{i=1} \sum_{j=-1}^{j=1} h_{old}(i,j) \quad . \tag{10}$$

If h(i,j) is an integer (which it should be), then Equation (8) illustrates that

Illiac-type communication [SIE85], integer addition, and integer division are tested with this algorithm.

4 Conclusion

The performance analysis and benchmarking of supercomputers is an area in which further investigation is necessary. This paper presents a methodology designed for the analysis of a specific parallel processing machine (the CM-2). The results of these tests should help characterize aspects of the performance of the CM-2, and in turn, aid in the future development of parallel processing computers.

Future work will include the simulation of other interconnection networks. Specifically, the PM2I network and the shuffle-exchange network will be further investigated. Both of these networks embody regular communication patterns which are frequently used in parallel algorithms. A more analytical analysis of the PM2I simulation will be conducted to better characterize the contention occurring in the CM-2. In addition, simulations of the shuffle network have been completed for both versions of the CM-2 software, and several 3-dimensional plots of the results have been made. More analysis of these results are needed. Finally, the floating point hardware was recently installed in the CM-2 at NASA Ames, and it will be interesting to determine what effects this has on the execution time of floating point operations.

5 References

[HIL85] Hillis, W. Daniel, *The Connection Machine*, The MIT Press, Cambridge, Massachusetts, 1985, 190pp.

[IEE85] *IEEE Standard for Binary Floating-Point Arithmetic*, The Institute of Electrical and Electronic Engineers, New York, 1985, 18pp.

[OFA87] O'Farrell, *Report on Instruction Timing for the Connection Machine (Model CM-1)*, Syracuse Technical Report 87-1, Syracuse University, Syracuse, New York, 1987.

[SIE85] Siegel, H.J., *Interconnection Networks for Large-Scale Parallel Processing*, D.C. Heath and Company, Lexington, Massachusetts, 1985, pp. 23-24.

[TFC86] T. F. Chan and Y. Saad, "Multigrid Algorithms on the Hypercube Multiprocessor," *IEEE Transactions on Computers*, November 1986.

[TM86] *Introduction to Data Level Parallelism*, Technical Report 86.14, Thinking Machines Corp., Cambridge, Massachusetts, April 1986, 60pp.

[TM87] *Model CM-2 Technical Summary*, Technical Report HA87-4, Thinking Machines Corp., Cambridge, Massachusetts, April 1987, 60 pp.

[TM87b] Connection Machine System Software Documentation, five volumes: *Connection Machine System Overview, Connection Machine System Front End: VAX ULTRIX, Connection Machine Front End: Symbolics, Connection Machine Programming in C*, Connection Machine Programming in *Lisp.*

[HWA79] *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley & Sons, New York, 1979, pp. 75-76.

APPENDICES

Appendix A

Performance Measurement Skeleton

```
(defvar *data*)
(defvar *fp*)

(defun enable-all ()
 (cm:move-constant-always cm:context-flag 1 1)
 (cm:move-constant-always cm:overflow-flag 0 1))

(*defun concentration (address-pvar)
 (declare (type (field-pvar cm:*cube-address-length*) address-pvar))
 (*let ((rcvd-count))
  (declare (type (field-pvar 16) rcvd-count))
  (*all
   (*set rcvd-count (!! (the fixnum 0))))
  (*pset :add (!! (the fixnum 1)) rcvd-count address-pvar)
  (let ((n-a-p (cm:global-count cm:context-flag)))
   (format *fp* "   number of active processors: ~D~%" n-a-p)
   (*all
    (*when (/=!! rcvd-count (!! (the fixnum 0)))
     (let* ((n-receiving (cm:global-count cm:context-flag))
            (total-sum (*sum rcvd-count))
            (max-rcvd (*max rcvd-count))
            (av-received (/ (float total-sum) (float n-receiving))))
      (format *fp* "   number receiving: ~D (~,1F%)~%" n-receiving
              (* 100.0 (/ n-receiving (float n-a-p))))
      (format *fp* "   average received: ~,2F~%" av-received)
      (format *fp* "   max received: ~D~%" max-rcvd)))))))

(defun statistics (values time-loop)
 (let ((average   0)
       (std-dev   0)
       (sum       0)
       (sqr-sum   0)
       (maximum   0)
       (minimum   0)
       (temp      0))
  (dotimes (i time-loop)
   (setq temp (vector-pop values))
   (format *data* "~F~%" temp)
   (setq sum (+ sum temp))
   (setq sqr-sum (+ sqr-sum (* temp temp)))
   (setq maximum (max maximum temp))
   (if (= i 0)
    (setq minimum temp))
   (setq minimum (min minimum temp)))
  (setq average (/ sum time-loop))
  (setq std-dev (sqrt (- (/ sqr-sum (- time-loop 1))
                 (/ (* time-loop (* average average))(- time-loop 1)))))
  (format *fp* "   Average: ~f~%" average)
  (format *fp* "   Standard Deviation: ~f~%" std-dev)
  (format *fp* "   Maximum: ~f~%" maximum)
  (format *fp* "   Minimum: ~f~%" minimum)))
;;;
;;;  Timing #
;;;  Date
;;;  Written by:            David Myers
;;;  Description:  This is the basic skeleton used for the tests.
;;;  Active Processors:
;;;  Size of Data Used:
;;;
(*defun time-?? (time-loop test-loop)
 (format *fp* "~%SHORT DESCRIPTION~%")
 (format *data* "~%SHORT DESCRIPTION~%")
 (*all
  (let ((values 0))
   (setq values (make-array '(100) :fill-pointer 0))


   ;;
   ;; Test set-up area
   ;;

   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
```

```
          (cm:time
            (dotimes (j test-loop)

            ;;
            ;; Instruction to be timed
            ;;

          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
(defun main ()
  (setq *fp* (open "misc3-stats" :direction :output))
  (setq *data* (open "misc3-data" :direction :output))
  (time-?? 100 XXXX)
  (close *data*)
  (close *fp*))
```

Appendix B

Communication Tests

```
;;;
;;; Timing #1
;;; 6/29/88
;;; Written by:              David Myers
;;; Description:  This program tests the bidirectional communication
;;;               capabilities within the CM2.
;;; Active Processors:        PE# mod 16 = 0
;;; Size of data used:        32 bit unsigned integers
;;;
(*defun time-1 (time-loop test-loop)
 (format *fp* "time-1~%")
 (format *data* "time-1~%")
 (let ((values 0))
  (enable-all)
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((send (!! (the fixnum 2000)))
          (s-addr (!! (the fixnum 0)))
          (receive (!! (the fixnum 0))))
     (declare (type (field-pvar 32) send))
     (declare (type (field-pvar 32) receive))
     (declare (type (field-pvar cm:*cube-address-length*) s-addr))
     (*when (=!! (mod!! (the (pvar (unsigned-byte cm:*cube-address-length*))
               (self-address!!))(!! (the fixnum 16)))(!! (the fixnum 0)))
      (*set s-addr (logxor!! (the (pvar (unsigned-byte
            cm:*cube-address-length*))(self-address!!))(!! (the fixnum 16))))
      ;;
      ;; Instruction to be timed.
      ;;
      (dotimes (i time-loop)
       (format t "~d~%" i)
       (multiple-value-bind (a cm-time b c)
        (cm:time
         (dotimes (i test-loop)
         (with-paris-from-*lisp
         (cm:send (pvar-location receive) (pvar-location s-addr)
                (pvar-location send) 32)))
         :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
      (concentration s-addr)
      (statistics values time-loop))))))
;;;
;;; Timing #2
;;; 6/29/88
;;; Written by:              David Myers
;;; Description:  This program does the same test as Timing #1
;;;               except the communications are only one-way.
;;; Active Processors:        PE# mod 32 = 0
;;; Size of data used:        32 bit unsigned integers
;;;
(*defun time-2 (time-loop test-loop)
 (format *fp* "time-2~%")
 (format *data* "time-2~%")
 (let ((values 0))
  (enable-all)
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((send (!! (the fixnum 2000)))
          (s-addr (!! (the fixnum 0)))
          (receive (!! (the fixnum 0))))
     (declare (type (field-pvar 32) send))
     (declare (type (field-pvar 32) receive))
     (declare (type (field-pvar cm:*cube-address-length*) s-addr))
     (*when (=!! (mod!! (the (pvar (unsigned-byte cm:*cube-address-length*))
               (self-address!!))(!! (the fixnum 16)))(!! (the fixnum 0)))
      (*set s-addr (logxor!! (the (pvar (unsigned-byte
            cm:*cube-address-length*))(self-address!!))(!! (the fixnum 16))))
      ;;
      ;; Instruction to be timed.
      ;;
      (dotimes (i time-loop)
       (format t "~d~%" i)
       (multiple-value-bind (a cm-time b c)
```

```
          (cm:time
           (dotimes (i test-loop)
           (with-paris-from-*lisp
           (cm:send (pvar-location receive) (pvar-location s-addr)
                (pvar-location send) 32)))
          :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values)))
       (concentration s-addr)
       (statistics values time-loop))))))
;;;
;;; Timing #3
;;; Written by:             David Myers
;;; Description: This program reports the time is take to send
;;;              a message from PE 0 to PE cm:*user-cube-address-limit*
;;; Active Processor:        PE 0
;;; Size of data:           32 bit unsigned integers
;;;
(*defun time-3 (time-loop test-loop)
 (format *fp* "time-3~%")
 (format *data* "time-3~%")
 (let ((values 0))
  (enable-all)
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((send (!! (the fixnum 2000)))
       (s-addr (!! (the fixnum 0)))
       (receive (!! (the fixnum 0))))
    (declare (type (field-pvar 32) send))
    (declare (type (field-pvar 32) receive))
    (declare (type (field-pvar cm:*cube-address-length*) s-addr))
    (*when (=!! (the (pvar (unsigned-byte cm:*cube-address-length*))
            (self-address!!))(!! (the fixnum 0)))
     (*set s-addr (logxor!! (the (pvar (unsigned-byte
          cm:*cube-address-length*))(self-address!!))
          (!! (the fixnum (- cm:*user-cube-address-limit* 1)))))
     ;;
     ;; Instruction to be timed.
     ;;
     (dotimes (i time-loop)
       (format t "~d~%" i)
       (multiple-value-bind (a cm-time b c)
        (cm:time
         (dotimes (i test-loop)
         (with-paris-from-*lisp
         (cm:send (pvar-location receive) (pvar-location s-addr)
              (pvar-location send) 32)))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
     (concentration s-addr)
     (statistics values time-loop))))))
;;;
;;; Timing #20-2
;;; 7/18/88, 9/17/18
;;; Written by:          David Myers
;;; Description: This program reports the time it take the connection
;;;              machine to simulate the shuffle-exchange from code
;;;              written in *lisp. The calculation of the destination
;;;              address is not included in this timing.
;;; Active Processors:   All
;;; Size of Data Used:   2 bit unisigned integers
;;;
(*defun time-20-2 (time-loop test-loop)
 (format *fp* "~%Timing 20: 2 bit msgs.~%")
 (format *data* "~%Timing 20: 2 bit msgs.~%")
 (let ((values 0)
     (index (/ test-loop 10)))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((dest-address (!! (the fixnum 0)))
       (temp (the (field-pvar cm:*cube-address-length*)
                (self-address!!)))
       (data-sent (!! (the fixnum 3)))
       (data-recvd (!! (the fixnum 3)))
```

```
      (number-pes (!! (the fixnum cm:*user-cube-address-limit*))))
    (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) dest-address))
    (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) temp))
    (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) number-pes))
    (declare (type (field-pvar 2) data-sent))
    (declare (type (field-pvar 2) data-recvd))
    (dotimes (k (- cm:*cube-address-length* 1))
     (format *fp* "  ~d way shuffle.~%" (+ k 1))
     (dotimes (calc (+ k 1))
      (*if (<!! temp (/!! number-pes (!! (the fixnum 2))))
       (*set temp (mod!! (*!! temp (!! (the fixnum 2))) number-pes))
       (*set temp (mod!! (+!! (!! (the fixnum 1))(*!! temp
                (!! (the fixnum 2)))) number-pes))))
     (*set dest-address temp)
     (dotimes (i time-loop)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j index)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
     (concentration dest-address)
     (statistics values time-loop)
     (*set temp (the (field-pvar cm:*cube-address-length*)
               (self-address!!))))))))
;;;
;;; Timing #20-4
;;; 7/18/88, 9/17/18
;;; Written by:         David Myers
;;; Description:  This program reports the time it take the connection
;;;             machine to simulate the shuffle-exchange from code
;;;             written in *lisp.  The calculation of the destination
;;;             address is not included in this timing.
;;; Active Processors:      All
;;; Size of Data Used:      4 bit unisigned integers
;;;
(*defun time-20-4 (time-loop test-loop)
 (format *fp* "~%Timing 20: 4 bit msgs.~%")
 (format *data* "~%Timing 20: 4 bit msgs.~%")
 (let ((values 0)
     (index (/ test-loop 10)))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((dest-address (!! (the fixnum 0)))
       (temp (the (field-pvar cm:*cube-address-length*)
               (self-address!!)))
       (data-sent (!! (the fixnum 15)))
       (data-recvd (!! (the fixnum 15)))
       (number-pes (!! (the fixnum cm:*user-cube-address-limit*))))
    (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) dest-address))
    (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) temp))
    (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) number-pes))
    (declare (type (field-pvar 4) data-sent))
    (declare (type (field-pvar 4) data-recvd))
    (dotimes (k (- cm:*cube-address-length* 1))
     (format *fp* "  ~d way shuffle.~%" (+ k 1))
     (dotimes (calc (+ k 1))
      (*if (<!! temp (/!! number-pes (!! (the fixnum 2))))
       (*set temp (mod!! (*!! temp (!! (the fixnum 2))) number-pes))
       (*set temp (mod!! (+!! (!! (the fixnum 1))(*!! temp
                (!! (the fixnum 2)))) number-pes))))
     (*set dest-address temp)
     (dotimes (i time-loop)
      (multiple-value-bind (a cm-time b c)
```

```
          (cm:time
           (dotimes (j index)
            (*pset :no-collisions data-sent data-recvd dest-address)
            (*pset :no-collisions data-sent data-recvd dest-address)
            (*pset :no-collisions data-sent data-recvd dest-address)
            (*pset :no-collisions data-sent data-recvd dest-address)
            (*pset :no-collisions data-sent data-recvd dest-address)
            (*pset :no-collisions data-sent data-recvd dest-address)
            (*pset :no-collisions data-sent data-recvd dest-address)
            (*pset :no-collisions data-sent data-recvd dest-address)
            (*pset :no-collisions data-sent data-recvd dest-address)
            (*pset :no-collisions data-sent data-recvd dest-address))
           :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values)))
        (concentration dest-address)
        (statistics values time-loop)
        (*set temp (the (field-pvar cm:*cube-address-length*)
                        (self-address!!)))))))
;;;
;;; Timing #20-8
;;; 7/18/88, 9/17/18
;;; Written by:         David Myers
;;; Description:  This program reports the time it take the connection
;;;               machine to simulate the shuffle-exchange from code
;;;               written in *lisp.  The calculation of the destination
;;;               address is not included in this timing.
;;; Active Processors:    All
;;; Size of Data Used:    8 bit unisigned integers
;;;
(*defun time-20-8 (time-loop test-loop)
 (format *fp* "~%Timing 20: 8 bit msgs.~%")
 (format *data* "~%Timing 20: 8 bit msgs.~%")
 (let ((values 0)
       (index (/ test-loop 10)))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((dest-address (!! (the fixnum 0)))
          (temp (the (field-pvar cm:*cube-address-length*)
                     (self-address!!)))
          (data-sent (random!! (!! (the fixnum (ash 1 8)))))
          (data-recvd (random!! (!! (the fixnum (ash 1 8)))))
          (number-pes (!! (the fixnum cm:*user-cube-address-limit*))))
      (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) dest-address))
      (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) temp))
      (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) number-pes))
      (declare (type (field-pvar 8) data-sent))
      (declare (type (field-pvar 8) data-recvd))
      (dotimes (k (- cm:*cube-address-length* 1))
       (format *fp* "  ~d way shuffle.~%" (+ k 1))
       (dotimes (calc (+ k 1))
        (*if (<!! temp (/!! number-pes (!! (the fixnum 2))))
         (*set temp (mod!! (*!! temp (!! (the fixnum 2))) number-pes))
         (*set temp (mod!! (+!! (!! (the fixnum 1))(*!! temp
                   (!! (the fixnum 2)))) number-pes))))
       (*set dest-address temp)
       (dotimes (i time-loop)
        (multiple-value-bind (a cm-time b c)
         (cm:time
          (dotimes (j index)
           (*pset :no-collisions data-sent data-recvd dest-address)
           (*pset :no-collisions data-sent data-recvd dest-address)
           (*pset :no-collisions data-sent data-recvd dest-address)
           (*pset :no-collisions data-sent data-recvd dest-address)
           (*pset :no-collisions data-sent data-recvd dest-address)
           (*pset :no-collisions data-sent data-recvd dest-address)
           (*pset :no-collisions data-sent data-recvd dest-address)
           (*pset :no-collisions data-sent data-recvd dest-address)
           (*pset :no-collisions data-sent data-recvd dest-address)
           (*pset :no-collisions data-sent data-recvd dest-address))
          :return-statistics-only-p t)
         (vector-push (/ cm-time test-loop) values)))
       (concentration dest-address)
       (statistics values time-loop)
```

```
      (*set temp (the (field-pvar cm:*cube-address-length*)
                   (self-address!!))))))))
;;;
;;; Timing #20-16
;;; 7/18/88, 9/17/18
;;; Written by:         David Myers
;;; Description:  This program reports the time it take the connection
;;;               machine to simulate the shuffle-exchange from code
;;;               written in *lisp.  The calculation of the destination
;;;               address is not included in this timing.
;;; Active Processors:      All
;;; Size of Data Used:      16 bit unisigned integers
;;;
(*defun time-20-16 (time-loop test-loop)
 (format *fp* "~%Timing 20: 16 bit msgs.~%")
 (format *data* "~%Timing 20: 16 bit msgs.~%")
 (let ((values 0)
      (index (/ test-loop 10)))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((dest-address (!! (the fixnum 0)))
        (temp (the (field-pvar cm:*cube-address-length*)
                    (self-address!!)))
        (data-sent (random!! (!! (the fixnum (ash 1 16)))))
        (data-recvd (random!! (!! (the fixnum (ash 1 16)))))
        (number-pes (!! (the fixnum cm:*user-cube-address-limit*))))
    (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) dest-address))
    (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) temp))
    (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) number-pes))
    (declare (type (field-pvar 16) data-sent))
    (declare (type (field-pvar 16) data-recvd))
    (dotimes (k (- cm:*cube-address-length* 1))
     (format *fp* "  ~d way shuffle.~%" (+ k 1))
     (dotimes (calc (+ k 1))
      (*if (<!! temp (/!! number-pes (!! (the fixnum 2))))
       (*set temp (mod!! (*!! temp (!! (the fixnum 2))) number-pes))
       (*set temp (mod!! (+!! (!! (the fixnum 1))(*!! temp
                 (!! (the fixnum 2)))) number-pes))))
     (*set dest-address temp)
     (dotimes (i time-loop)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j index)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address)
         (*pset :no-collisions data-sent data-recvd dest-address))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
     (concentration dest-address)
     (statistics values time-loop)
     (*set temp (the (field-pvar cm:*cube-address-length*)
                  (self-address!!))))))))
;;;
;;; Timing #20-32
;;; 7/18/88, 9/17/18
;;; Written by:         David Myers
;;; Description:  This program reports the time it take the connection
;;;               machine to simulate the shuffle-exchange from code
;;;               written in *lisp.  The calculation of the destination
;;;               address is not included in this timing.
;;; Active Processors:      All
;;; Size of Data Used:      32 bit unisigned integers
;;;
(*defun time-20-32 (time-loop test-loop)
 (format *fp* "~%Timing 20: 32 bit msgs.~%")
 (format *data* "~%Timing 20: 32 bit msgs.~%")
```

```
(let ((values 0)
     (index (/ test-loop 10)))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((dest-address (!! (the fixnum 0)))
          (temp (the (field-pvar cm:*cube-address-length*)
                     (self-address!!)))
          (data-sent (random!! (!! (the fixnum (ash 1 32)))))
          (data-recvd (random!! (!! (the fixnum (ash 1 32)))))
          (number-pes (!! (the fixnum cm:*user-cube-address-limit*))))
     (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) dest-address))
     (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) temp))
     (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) number-pes))
     (declare (type (field-pvar 32) data-sent))
     (declare (type (field-pvar 32) data-recvd))
     (dotimes (k (- cm:*cube-address-length* 1))
      (format *fp* "  ~d way shuffle.~%" (+ k 1))
      (dotimes (calc (+ k 1))
       (*if (<!! temp (/!! number-pes (!! (the fixnum 2))))
        (*set temp (mod!! (*!! temp (!! (the fixnum 2))) number-pes))
        (*set temp (mod!! (+!! (!! (the fixnum 1))(*!! temp
                       (!! (the fixnum 2)))) number-pes))))
      (*set dest-address temp)
      (dotimes (i time-loop)
       (multiple-value-bind (a cm-time b c)
        (cm:time
         (dotimes (j index)
          (*pset :no-collisions data-sent data-recvd dest-address)
          (*pset :no-collisions data-sent data-recvd dest-address)
          (*pset :no-collisions data-sent data-recvd dest-address)
          (*pset :no-collisions data-sent data-recvd dest-address)
          (*pset :no-collisions data-sent data-recvd dest-address)
          (*pset :no-collisions data-sent data-recvd dest-address)
          (*pset :no-collisions data-sent data-recvd dest-address)
          (*pset :no-collisions data-sent data-recvd dest-address)
          (*pset :no-collisions data-sent data-recvd dest-address)
          (*pset :no-collisions data-sent data-recvd dest-address))
         :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)
      (*set temp (the (field-pvar cm:*cube-address-length*)
                  (self-address!!))))))))
;;;
;;; Timing #20-64
;;; 7/18/88, 9/17/18
;;; Written by:          David Myers
;;; Description: This program reports the time it take the connection
;;;              machine to simulate the shuffle-exchange from code
;;;              written in *lisp.  The calculation of the destination
;;;              address is not included in this timing.
;;; Active Processors:   All
;;; Size of Data Used:   64 bit unisigned integers
;;;
(*defun time-20-64 (time-loop test-loop)
 (format *fp* "~%Timing 20: 64 bit msgs.~%")
 (format *data* "~%Timing 20: 64 bit msgs.~%")
 (let ((values 0)
      (index (/ test-loop 10)))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((dest-address (!! (the fixnum 0)))
          (temp (the (field-pvar cm:*cube-address-length*)
                     (self-address!!)))
          (data-sent (random!! (!! (the fixnum (ash 1 64)))))
          (data-recvd (random!! (!! (the fixnum (ash 1 64)))))
          (number-pes (!! (the fixnum cm:*user-cube-address-limit*))))
     (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) dest-address))
     (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) temp))
     (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) number-pes))
     (declare (type (field-pvar 64) data-sent))
     (declare (type (field-pvar 64) data-recvd))
     (dotimes (k (- cm:*cube-address-length* 1))
```

```
        (format *fp* "  ~d way shuffle.~%" (+ k 1))
        (dotimes (calc (+ k 1))
          (*if (<!! temp (/!! number-pes (!! (the fixnum 2))))
            (*set temp (mod!! (*!! temp (!! (the fixnum 2))) number-pes))
            (*set temp (mod!! (+!! (!! (the fixnum 1))(*!! temp
                        (!! (the fixnum 2)))) number-pes))))
        (*set dest-address temp)
        (dotimes (i time-loop)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (dotimes (j index)
                (*pset :no-collisions data-sent data-recvd dest-address)
                (*pset :no-collisions data-sent data-recvd dest-address)
                (*pset :no-collisions data-sent data-recvd dest-address)
                (*pset :no-collisions data-sent data-recvd dest-address)
                (*pset :no-collisions data-sent data-recvd dest-address)
                (*pset :no-collisions data-sent data-recvd dest-address)
                (*pset :no-collisions data-sent data-recvd dest-address)
                (*pset :no-collisions data-sent data-recvd dest-address)
                (*pset :no-collisions data-sent data-recvd dest-address)
                (*pset :no-collisions data-sent data-recvd dest-address))
              :return-statistics-only-p t)
            (vector-push (/ cm-time test-loop) values)))
        (concentration dest-address)
        (statistics values time-loop)
        (*set temp (the (field-pvar cm:*cube-address-length*)
                    (self-address!!))))))))
;;;
;;; Timing #20-80
;;; 7/18/88, 9/17/18
;;; Written by:        David Myers
;;; Description: This program reports the time it take the connection
;;;          machine to simulate the shuffle-exchange from code
;;;          written in *lisp. The calculation of the destination
;;;          address is not included in this timing.
;;; Active Processors:    All
;;; Size of Data Used:    80 bit unisigned integers
;;;
(*defun time-20-80 (time-loop test-loop)
  (format *fp* "~%Timing 20: 80 bit msgs.~%")
  (format *data* "~%Timing 20: 80 bit msgs.~%")
  (let ((values 0)
        (index (/ test-loop 10)))
    (setq values (make-array '(100) :fill-pointer 0))
    (*all
      (*let ((dest-address (!! (the fixnum 0)))
             (temp (the (field-pvar cm:*cube-address-length*)
                    (self-address!!)))
             (data-sent (random!! (!! (the fixnum (ash 1 80)))))
             (data-recvd (random!! (!! (the fixnum (ash 1 80)))))
             (number-pes (!! (the fixnum cm:*user-cube-address-limit*))))
        (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) dest-address))
        (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) temp))
        (declare (type (field-pvar (+ 1 cm:*cube-address-length*)) number-pes))
        (declare (type (field-pvar 80) data-sent))
        (declare (type (field-pvar 80) data-recvd))
        (dotimes (k (- cm:*cube-address-length* 1))
          (format *fp* "  ~d way shuffle.~%" (+ k 1))
          (dotimes (calc (+ k 1))
            (*if (<!! temp (/!! number-pes (!! (the fixnum 2))))
              (*set temp (mod!! (*!! temp (!! (the fixnum 2))) number-pes))
              (*set temp (mod!! (+!! (!! (the fixnum 1))(*!! temp
                          (!! (the fixnum 2)))) number-pes))))
          (*set dest-address temp)
          (dotimes (i time-loop)
            (multiple-value-bind (a cm-time b c)
              (cm:time
                (dotimes (j index)
                  (*pset :no-collisions data-sent data-recvd dest-address)
                  (*pset :no-collisions data-sent data-recvd dest-address)
                  (*pset :no-collisions data-sent data-recvd dest-address)
                  (*pset :no-collisions data-sent data-recvd dest-address)
                  (*pset :no-collisions data-sent data-recvd dest-address)
```

```
              (*pset :no-collisions data-sent data-recvd dest-address)
              (*pset :no-collisions data-sent data-recvd dest-address)
              (*pset :no-collisions data-sent data-recvd dest-address)
              (*pset :no-collisions data-sent data-recvd dest-address)
              (*pset :no-collisions data-sent data-recvd dest-address))
             :return-statistics-only-p t)
            (vector-push (/ cm-time test-loop) values)))
          (concentration dest-address)
          (statistics values time-loop)
          (*set temp (the (field-pvar cm:*cube-address-length*)
                      (self-address!!))))))))
;;;
;;; Timing #21
;;; 7/18/88
;;; Written by:          David Myers
;;; Description:  This program reports the time it take the connection
;;;               machine to simulate the shuffle-exchange from code
;;;               written in *lisp.  The calculation of the destination
;;;               address is included in this timing.
;;; Active Processors:    All
;;; Size of Data Used:    32 bit unisigned integers
;;;
(*defun time-21 (time-loop test-loop)
 (format *fp* "~%Timing 21: perfect shuffle~%")
 (format *data* "~%Timing 21: perfect shuffle~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((dest-address (!! (the fixnum 0)))
          (data-sent (random!! (!! (the fixnum (ash 1 32)))))
          (data-recvd (random!! (!! (the fixnum (ash 1 32)))))
          (number-pes (!! (the fixnum cm:*user-cube-address-limit*))))
     (declare (type (field-pvar cm:*cube-address-length*) dest-address))
     (declare (type (field-pvar cm:*cube-address-length*) number-pes))
     (declare (type (field-pvar 32) data-sent))
     (declare (type (field-pvar 32) data-recvd))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
         (*when (<!! (the (field-pvar cm:*cube-address-length*)
                      (self-address!!))(/!! number-pes (!! (the fixnum 2))))
          (*set dest-address (mod!! (*!!
                      (the (field-pvar cm:*cube-address-length*)
                      (self-address!!)) (!! (the fixnum 2))) number-pes)))
         (*when (>=!! (the (field-pvar cm:*cube-address-length*)
                      (self-address!!))(/!! number-pes (!! (the fixnum 2))))
          (*set dest-address (mod!! (+!! (!! (the fixnum 1))(*!!
                      (the (field-pvar cm:*cube-address-length*)
                      (self-address!!)) (!! (the fixnum 2)))) number-pes)))
         (*pset :no-collisions data-sent data-recvd dest-address))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
     (concentration dest-address)
     (statistics values time-loop)))))
;;;
;;; Timing #22
;;; 7/18/88
;;; Written by:          David Myers
;;; Description:  This program reports the time it take the connection
;;;               machine to simulate the shuffle-exchange from code
;;;               written in *lisp.  The calculation of the destination address
;;;               is done by bit shifting, and the calculation of the
;;;               destination address is not included in the time.
;;; Active Processors:    All
;;; Size of Data Used:    32 bit unisigned integers
;;;
(*defun time-22 (time-loop test-loop)
 (format *fp* "~%Timing 22: perfect shuffle - bit shift~%")
 (format *data* "~%Timing 22: perfect shuffle - bit shift~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
```

```
(*all
 (*let ((dest-address (the (field-pvar cm:*cube-address-length*)
                           (self-address!!)))
       (data-sent (random!! (!! (the fixnum (ash 1 32)))))
       (data-recvd (random!! (!! (the fixnum (ash 1 32)))))
       (shift-amt (!! (the fixnum 1)))
       (number-pes (!! (the fixnum cm:*user-cube-address-limit*))))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (declare (type (field-pvar cm:*cube-address-length*) number-pes))
   (declare (type (field-pvar 32) data-sent))
   (declare (type (field-pvar 32) data-recvd))
   (declare (type (signed-pvar 2) shift-amt))
   (with-paris-from-*lisp
    (cm:unsigned-shift (pvar-location dest-address)(pvar-location shift-amt)
                (pvar-length dest-address)(pvar-length shift-amt)))
   (*when (eq!! (the (pvar boolean)(*lisp-i::overflow-flag!!))
                         (the (pvar boolean) t!!))
    (*set dest-address (logxor!! dest-address (!! (the fixnum 1)))))
   (format *fp* "processor 2000: ~D~%" (pref dest-address 2000))
   (format *fp* "processor 7000: ~D~%" (pref dest-address 7000))
   (format *fp* "processor 8191: ~D~%" (pref dest-address 8191))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (*pset :no-collisions data-sent data-recvd dest-address))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop)))))
;;;
;;; Timing #23
;;; 7/18/88
;;; Written by:        David Myers
;;; Description:  This program reports the time it take the connection
;;;              machine to simulate the shuffle-exchange from code
;;;              written in *lisp.  The calculation of the destination address
;;;              is done by bit shifting, and the calculation of the
;;;              destination address is included in the time.
;;; Active Processors:     All
;;; Size of Data Used:     32 bit unisigned integers
;;;
(*defun time-23 (time-loop test-loop)
 (format *fp* "~%Timing 23: perfect shuffle - bit shift - calc. incl.~%")
 (format *data* "~%Timing 23: perfect shuffle - bit shift - calc. incl.~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((dest-address (the (field-pvar cm:*cube-address-length*)
                             (self-address!!)))
         (data-sent (random!! (!! (the fixnum (ash 1 32)))))
         (data-recvd (random!! (!! (the fixnum (ash 1 32)))))
         (shift-amt (!! (the fixnum 1)))
         (number-pes (!! (the fixnum cm:*user-cube-address-limit*))))
     (declare (type (field-pvar cm:*cube-address-length*) dest-address))
     (declare (type (field-pvar cm:*cube-address-length*) number-pes))
     (declare (type (field-pvar 32) data-sent))
     (declare (type (field-pvar 32) data-recvd))
     (declare (type (signed-pvar 2) shift-amt))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
         (with-paris-from-*lisp
          (cm:unsigned-shift (pvar-location dest-address)
                 (pvar-location shift-amt)(pvar-length dest-address)
                 (pvar-length shift-amt)))
         (*when (eq!! (the (pvar boolean)(*lisp-i::overflow-flag!!))
                 (the (pvar boolean) t!!))
          (*set dest-address (logxor!! dest-address (!! (the fixnum 1)))))
         (*pset :no-collisions data-sent data-recvd dest-address))
```

```
               :return-statistics-only-p t)
             (*set dest-address (the (field-pvar cm:*cube-address-length*)
                        (self-address!!)))
             (vector-push (/ cm-time test-loop) values)))
          (concentration dest-address)
          (statistics values time-loop)))))
;;;
;;; Timing #24
;;; 7/18/88
;;; Written by:         David Myers
;;; Description:  This program reports the time it take the connection
;;;            machine to simulate the shuffle-exchange from code
;;;            written in *lisp.  The calculation of the destination address
;;;            is done by bit shifting, and the calculation of the
;;;            destination address is included in the time.  Timing #24
;;;            differs from timing #23 in that the number one "1" used
;;;            in the calculation of the destination address is not
;;;            broadcast during the calculation.  Instead, the broadcasting
;;;            is done upon initialisation of the variable "one".
;;; Active Processors:     All
;;; Size of Data Used:     32 bit unisigned integers
;;;
(*defun time-24 (time-loop test-loop)
  (format *fp* "~%Timing 24: perfect shuffle - bit shift - calc. incl. - 1 bdcst~%")
  (format *data* "~%Timing 24: perfect shffl - bit shift - calc. incl - 1 bdcst.~%")
  (let ((values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*all
      (*let ((dest-address (the (field-pvar cm:*cube-address-length*)
                                    (self-address!!)))
             (data-sent (random!! (!! (the fixnum (ash 1 32)))))
             (data-recvd (random!! (!! (the fixnum (ash 1 32)))))
             (shift-amt (!! (the fixnum 1)))
             (one (!! (the fixnum 1)))
             (number-pes (!! (the fixnum cm:*user-cube-address-limit*))))
        (declare (type (field-pvar cm:*cube-address-length*) dest-address))
        (declare (type (field-pvar cm:*cube-address-length*) one))
        (declare (type (field-pvar cm:*cube-address-length*) number-pes))
        (declare (type (field-pvar 32) data-sent))
        (declare (type (field-pvar 32) data-recvd))
        (declare (type (signed-pvar 2) shift-amt))
        (dotimes (i time-loop)
          (format t "~d~%" i)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (dotimes (j test-loop)
                (with-paris-from-*lisp
                  (cm:unsigned-shift (pvar-location dest-address)
                        (pvar-location shift-amt)
                        (pvar-length dest-address)(pvar-length shift-amt)))
                (*when (eq!! (the (pvar boolean)(*lisp-i::overflow-flag!!))
                        (the (pvar boolean) t!!))
                  (*set dest-address (logxor!! dest-address one)))
                (*pset :no-collisions data-sent data-recvd dest-address))
              :return-statistics-only-p t)
            (*set dest-address (the (field-pvar cm:*cube-address-length*)
                        (self-address!!)))
            (vector-push (/ cm-time test-loop) values)))
        (concentration dest-address)
        (statistics values time-loop)))))
;;;
;;; Timing #25
;;; 7/23/88
;;; Written by:         David Myers
;;; Description:  This program tests the time required for PEs to get information
;;;            from their four nearest neighbors.  The NEWS operation is
;;;            is used, and values received are stored.  Border PEs are
;;;            not members of the active set.
;;; Active Processors:     All
;;; Size of Data Used:     32 bit unisigned integers
;;;
(*defun time-25 (time-loop test-loop)
  (format *fp* "~%Timing 25: NEWS grid - 4 nearest neighbors~%")
```

```
(format *data* "~%Timing 25: NEWS grid - 4 nearest neighbors~%")
(let ((values 0))
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((x-addr (!! (the fixnum 0)))
        (y-addr (!! (the fixnum 0)))
        (a (random!! (!! (the fixnum (ash 1 32)))))
        (data-recvd (random!! (!! (the fixnum (ash 1 32))))))
    (declare (type (field-pvar cm:*physical-x-dimension-limit*) x-addr))
    (declare (type (field-pvar cm:*physical-y-dimension-limit*) y-addr))
    (declare (type (field-pvar 32) a))
    (declare (type (field-pvar 32) data-recvd))
    (with-paris-from-*lisp
     (cm:my-x-address (pvar-location x-addr))
     (cm:my-y-address (pvar-location y-addr)))
    (*when (and!! (/=!! x-addr (!! (the fixnum 0))
                   (!! (the fixnum (- cm:*physical-x-dimension-limit* 1))))
              (/=!! y-addr (!! (the fixnum 0))
                  (!! (the fixnum (- cm:*physical-y-dimension-limit* 1)))))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
         (*set data-recvd (pref-grid-relative!! a (!! (the fixnum -1))
                                  (!! (the fixnum 0))))
         (*set data-recvd  (pref-grid-relative!! a (!! (the fixnum 1))
                                  (!! (the fixnum 0))))
         (*set data-recvd  (pref-grid-relative!! a (!! (the fixnum 0))
                                  (!! (the fixnum -1))))
         (*set data-recvd  (pref-grid-relative!! a (!! (the fixnum 0))
                                  (!! (the fixnum 1)))))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (statistics values time-loop))))))
;;;
;;; Timing #26
;;; 7/23/88
;;; Written by:           David Myers
;;; Description:  This program tests the time required for PEs to get information
;;;               from their eight nearest neighbors. The NEWS operation is
;;;               is used, and values received are stored. Border PEs are
;;;               not members of the active set.
;;; Active Processors:    All
;;; Size of Data Used:    32 bit unisigned integers
;;;
(*defun time-26 (time-loop test-loop)
 (format *fp* "~%Timing 26: NEWS grid - 8 nearest neighbors~%")
 (format *data* "~%Timing 26: NEWS grid - 8 nearest neighbors~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((x-addr (!! (the fixnum 0)))
         (y-addr (!! (the fixnum 0)))
         (a (random!! (!! (the fixnum (ash 1 32)))))
         (data-recvd (random!! (!! (the fixnum (ash 1 32))))))
    (declare (type (field-pvar cm:*physical-x-dimension-limit*) x-addr))
    (declare (type (field-pvar cm:*physical-y-dimension-limit*) y-addr))
    (declare (type (field-pvar 32) a))
    (declare (type (field-pvar 32) data-recvd))
    (with-paris-from-*lisp
     (cm:my-x-address (pvar-location x-addr))
     (cm:my-y-address (pvar-location y-addr)))
    (*when (and!! (/=!! x-addr (!! (the fixnum 0))
                   (!! (the fixnum (- cm:*physical-x-dimension-limit* 1))))
              (/=!! y-addr (!! (the fixnum 0))
                  (!! (the fixnum (- cm:*physical-y-dimension-limit* 1)))))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
         (*set data-recvd (pref-grid-relative!! a (!! (the fixnum -1))
```

```
                                (!! (the fixnum 0))))
                (*set data-recvd  (pref-grid-relative!! a (!! (the fixnum 1))
                                (!! (the fixnum 0))))
                (*set data-recvd  (pref-grid-relative!! a (!! (the fixnum 0))
                                (!! (the fixnum -1))))
                (*set data-recvd  (pref-grid-relative!! a (!! (the fixnum 0))
                                (!! (the fixnum 1))))
                (*set data-recvd  (pref-grid-relative!! a (!! (the fixnum 1))
                                (!! (the fixnum 1))))
                (*set data-recvd  (pref-grid-relative!! a (!! (the fixnum 1))
                                (!! (the fixnum -1))))
                (*set data-recvd  (pref-grid-relative!! a (!! (the fixnum -1))
                                (!! (the fixnum -1))))
                (*set data-recvd  (pref-grid-relative!! a (!! (the fixnum -1))
                                (!! (the fixnum 1)))))
            :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values)))
      (statistics values time-loop))))))
;;;
;;; Timing #28-2
;;; 9/03/88
;;; Written by:          David Myers
;;; Description:  This program reports the time it takes the CM-2 to
;;;               simulate the pm2i network.
;;; Active Processors:    All
;;; Size of Data Used:    2 bit unisigned integers
;;;
(*defun time-28-2 (time-loop test-loop)
  (format *fp* "~%Timing 28-2: pm2i - 2 bit~%")
  (format *data* "~%Timing 28-2: pm2i - 2~%")
  (let ((values 0))
    (setq values (make-array '(10) :fill-pointer 0))
    (*all
     (*let ((dest-addr (the (pvar (unsigned-byte cm:*cube-address-length*))
                    (self-address!!)))
            (data-sent (random!! (!! (the fixnum 3))))
            (data-recvd (!! (the fixnum 0))))
       (declare (type (field-pvar cm:*cube-address-length*) dest-addr))
       (declare (type (field-pvar 2) data-sent))
       (declare (type (field-pvar 2) data-recvd))
       (dotimes (k cm:*cube-address-length*)
         (format *fp* "i = ~D~%" k)
         (*set dest-addr (+!! (!! (the fixnum (expt 2 k)))
           (the (pvar (unsigned-byte cm:*cube-address-length*))(self-address!!))))
         (dotimes (i time-loop)
           (format t "~d~%" i)
           (multiple-value-bind (a cm-time b c)
             (cm:time
              (dotimes (j test-loop)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr))
             :return-statistics-only-p t)
           (vector-push (/ cm-time (* test-loop 10.0)) values)))
       (statistics values time-loop))))))
;;;
;;; Timing #28-4
;;; 9/03/88
;;; Written by:          David Myers
;;; Description:  This program reports the time it takes the CM-2 to
;;;               simulate the pm2i network.
;;; Active Processors:    All
;;; Size of Data Used:    4 bit unisigned integers
;;;
(*defun time-28-4 (time-loop test-loop)
  (format *fp* "~%Timing 28-4: pm2i - 4 bit~%")
```

```
(format *data* "~%Timing 28-4: pm2i - 4~%")
(let ((values 0))
 (setq values (make-array '(10) :fill-pointer 0))
 (*all
  (*let ((dest-addr (the (pvar (unsigned-byte cm:*cube-address-length*))
              (self-address!!)))
      (data-sent (random!! (!! (the fixnum 15))))
      (data-recvd (!! (the fixnum 0))))
   (declare (type (field-pvar cm:*cube-address-length*) dest-addr))
   (declare (type (field-pvar 4) data-sent))
   (declare (type (field-pvar 4) data-recvd))
   (dotimes (k cm:*cube-address-length*)
    (format *fp* "i = ~D~%" k)
    (*set dest-addr (+!! (!! (the fixnum (expt 2 k)))
      (the (pvar (unsigned-byte cm:*cube-address-length*))(self-address!!))))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr))
      :return-statistics-only-p t)
     (vector-push (/ cm-time (* test-loop 10.0)) values)))
   (statistics values time-loop))))))
;;;
;;; Timing #28-8
;;; 9/03/88
;;; Written by:          David Myers
;;; Description:  This program reports the time it takes the CM-2 to
;;;               simulate the pm2i network.
;;; Active Processors:    All
;;; Size of Data Used:    8 bit unisigned integers
;;;
(*defun time-28-8 (time-loop test-loop)
 (format *fp* "~%Timing 28-8: pm2i - 8 bit~%")
 (format *data* "~%Timing 28-8: pm2i - 8~%")
 (let ((values 0))
 (setq values (make-array '(10) :fill-pointer 0))
 (*all
  (*let ((dest-addr (the (pvar (unsigned-byte cm:*cube-address-length*))
              (self-address!!)))
      (data-sent (random!! (!! (the fixnum (ash 1 8)))))
      (data-recvd (!! (the fixnum 0))))
   (declare (type (field-pvar cm:*cube-address-length*) dest-addr))
   (declare (type (field-pvar 8) data-sent))
   (declare (type (field-pvar 8) data-recvd))
   (dotimes (k cm:*cube-address-length*)
    (format *fp* "i = ~D~%" k)
    (*set dest-addr (+!! (!! (the fixnum (expt 2 k)))
      (the (pvar (unsigned-byte cm:*cube-address-length*))(self-address!!))))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
        (*pset :no-collisions data-sent data-recvd dest-addr)
```

```
            (*pset :no-collisions data-sent data-recvd dest-addr))
          :return-statistics-only-p t)
        (vector-push (/ cm-time (* test-loop 10.0)) values)))
      (statistics values time-loop))))))
;;;
;;; Timing #28-16
;;; 9/03/88
;;; Written by:           David Myers
;;; Description:  This program reports the time it takes the CM-2 to
;;;               simulate the pm2i network.
;;; Active Processors:    All
;;; Size of Data Used:    16 bit unisigned integers
;;;
(*defun time-28-16 (time-loop test-loop)
  (format *fp* "~%Timing 28-16: pm2i - 16 bit~%")
  (format *data* "~%Timing 28-16: pm2i - 16~%")
  (let ((values 0))
    (setq values (make-array '(10) :fill-pointer 0))
    (*all
     (*let ((dest-addr (the (pvar (unsigned-byte cm:*cube-address-length*))
                  (self-address!!)))
          (data-sent (random!! (!! (the fixnum (ash 1 16)))))
          (data-recvd (!! (the fixnum 0))))
       (declare (type (field-pvar cm:*cube-address-length*) dest-addr))
       (declare (type (field-pvar 16) data-sent))
       (declare (type (field-pvar 16) data-recvd))
       (dotimes (k cm:*cube-address-length*)
        (format *fp* "i = ~D~%" k)
        (*set dest-addr (+!! (!! (the fixnum (expt 2 k)))
          (the (pvar (unsigned-byte cm:*cube-address-length*))(self-address!!))))
        (dotimes (i time-loop)
         (format t "~d~%" i)
         (multiple-value-bind (a cm-time b c)
          (cm:time
           (dotimes (j test-loop)
            (*pset :no-collisions data-sent data-recvd dest-addr)
            (*pset :no-collisions data-sent data-recvd dest-addr)
            (*pset :no-collisions data-sent data-recvd dest-addr)
            (*pset :no-collisions data-sent data-recvd dest-addr)
            (*pset :no-collisions data-sent data-recvd dest-addr)
            (*pset :no-collisions data-sent data-recvd dest-addr)
            (*pset :no-collisions data-sent data-recvd dest-addr)
            (*pset :no-collisions data-sent data-recvd dest-addr)
            (*pset :no-collisions data-sent data-recvd dest-addr)
            (*pset :no-collisions data-sent data-recvd dest-addr))
          :return-statistics-only-p t)
        (vector-push (/ cm-time (* test-loop 10.0)) values)))
      (statistics values time-loop))))))
;;;
;;; Timing #28-32
;;; 9/03/88
;;; Written by:           David Myers
;;; Description:  This program reports the time it takes the CM-2 to
;;;               simulate the pm2i network.
;;; Active Processors:    All
;;; Size of Data Used:    32 bit unisigned integers
;;;
(*defun time-28-32 (time-loop test-loop)
  (format *fp* "~%Timing 28-32: pm2i - 32 bit~%")
  (format *data* "~%Timing 28-32: pm2i - 32~%")
  (let ((values 0))
    (setq values (make-array '(10) :fill-pointer 0))
    (*all
     (*let ((dest-addr (the (pvar (unsigned-byte cm:*cube-address-length*))
                  (self-address!!)))
          (data-sent (random!! (!! (the fixnum (ash 1 32)))))
          (data-recvd (!! (the fixnum 0))))
       (declare (type (field-pvar cm:*cube-address-length*) dest-addr))
       (declare (type (field-pvar 32) data-sent))
       (declare (type (field-pvar 32) data-recvd))
       (dotimes (k cm:*cube-address-length*)
        (format *fp* "i = ~D~%" k)
        (*set dest-addr (+!! (!! (the fixnum (expt 2 k)))
```

```
          (the (pvar (unsigned-byte cm:*cube-address-length*))(self-address!!))))
        (dotimes (i time-loop)
          (format t "~d~%" i)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (dotimes (j test-loop)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr))
            :return-statistics-only-p t)
          (vector-push (/ cm-time (* test-loop 10.0)) values)))
      (statistics values time-loop))))))
;;;
;;; Timing #28-64
;;; 9/03/88
;;; Written by:          David Myers
;;; Description:  This program reports the time it takes the CM-2 to
;;;               simulate the pm2i network.
;;; Active Processors:     All
;;; Size of Data Used:     64 bit unisigned integers
;;;
(*defun time-28-64 (time-loop test-loop)
 (format *fp* "~%Timing 28-64: pm2i - 64 bit~%")
 (format *data* "~%Timing 28-64: pm2i - 64~%")
 (let ((values 0))
  (setq values (make-array '(10) :fill-pointer 0))
  (*all
   (*let ((dest-addr (the (pvar (unsigned-byte cm:*cube-address-length*))
                  (self-address!!)))
        (data-sent (random!! (!! (the fixnum (ash 1 64)))))
        (data-recvd (!! (the fixnum 0))))
    (declare (type (field-pvar cm:*cube-address-length*) dest-addr))
    (declare (type (field-pvar 64) data-sent))
    (declare (type (field-pvar 64) data-recvd))
    (dotimes (k cm:*cube-address-length*)
      (format *fp* "i = ~D~%" k)
      (*set dest-addr (+!! (!! (the fixnum (expt 2 k)))
          (the (pvar (unsigned-byte cm:*cube-address-length*))(self-address!!))))
        (dotimes (i time-loop)
          (format t "~d~%" i)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (dotimes (j test-loop)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr)
                (*pset :no-collisions data-sent data-recvd dest-addr))
            :return-statistics-only-p t)
          (vector-push (/ cm-time (* test-loop 10.0)) values)))
      (statistics values time-loop))))))
;;;
;;; Timing #28-80
;;; 9/03/88
;;; Written by:          David Myers
;;; Description:  This program reports the time it takes the CM-2 to
;;;               simulate the pm2i network.
;;; Active Processors:     All
;;; Size of Data Used:     80 bit unisigned integers
;;;
(*defun time-28-80 (time-loop test-loop)
```

```
(format *fp* "~%Timing 28-80: pm2i - 80 bit~%")
(format *data* "~%Timing 28-80: pm2i - 80~%")
(let ((values 0))
 (setq values (make-array '(10) :fill-pointer 0))
 (*all
  (*let ((dest-addr (the (pvar (unsigned-byte cm:*cube-address-length*))
                   (self-address!!)))
        (data-sent (random!! (!! (the fixnum (ash 1 80)))))
        (data-recvd (!! (the fixnum 0))))
    (declare (type (field-pvar cm:*cube-address-length*) dest-addr))
    (declare (type (field-pvar 80) data-sent))
    (declare (type (field-pvar 80) data-recvd))
    (dotimes (k cm:*cube-address-length*)
     (format *fp* "i = ~D~%" k)
     (*set dest-addr (+!! (!! (the fixnum (expt 2 k)))
        (the (pvar (unsigned-byte cm:*cube-address-length*))(self-address!!))))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
         (*pset :no-collisions data-sent data-recvd dest-addr)
         (*pset :no-collisions data-sent data-recvd dest-addr)
         (*pset :no-collisions data-sent data-recvd dest-addr)
         (*pset :no-collisions data-sent data-recvd dest-addr)
         (*pset :no-collisions data-sent data-recvd dest-addr)
         (*pset :no-collisions data-sent data-recvd dest-addr)
         (*pset :no-collisions data-sent data-recvd dest-addr)
         (*pset :no-collisions data-sent data-recvd dest-addr)
         (*pset :no-collisions data-sent data-recvd dest-addr)
         (*pset :no-collisions data-sent data-recvd dest-addr))
       :return-statistics-only-p t)
      (vector-push (/ cm-time (* test-loop 10.0)) values)))
    (statistics values time-loop))))))
;;;
;;; Timing #28-128
;;; 9/03/88
;;; Written by:          David Myers
;;; Description:  This program reports the time it takes the CM-2 to
;;;               simulate the pm2i network.
;;; Active Processors:     All
;;; Size of Data Used:     128 bit unisigned integers
;;;
(*defun time-28-128 (time-loop test-loop)
 (format *fp* "~%Timing 28-128: pm2i - 128 bit~%")
 (format *data* "~%Timing 28-128: pm2i - 128~%")
 (let ((values 0))
  (setq values (make-array '(10) :fill-pointer 0))
  (*all
   (*let ((dest-addr (the (pvar (unsigned-byte cm:*cube-address-length*))
                    (self-address!!)))
         (data-sent (random!! (!! (the fixnum (ash 1 128)))))
         (data-recvd (!! (the fixnum 0))))
     (declare (type (field-pvar cm:*cube-address-length*) dest-addr))
     (declare (type (field-pvar 128) data-sent))
     (declare (type (field-pvar 128) data-recvd))
     (dotimes (k cm:*cube-address-length*)
      (format *fp* "i = ~D~%" k)
      (*set dest-addr (+!! (!! (the fixnum (expt 2 k)))
         (the (pvar (unsigned-byte cm:*cube-address-length*))(self-address!!))))
      (dotimes (i time-loop)
       (format t "~d~%" i)
       (multiple-value-bind (a cm-time b c)
        (cm:time
         (dotimes (j test-loop)
          (*pset :no-collisions data-sent data-recvd dest-addr)
          (*pset :no-collisions data-sent data-recvd dest-addr)
          (*pset :no-collisions data-sent data-recvd dest-addr)
          (*pset :no-collisions data-sent data-recvd dest-addr)
          (*pset :no-collisions data-sent data-recvd dest-addr)
          (*pset :no-collisions data-sent data-recvd dest-addr)
          (*pset :no-collisions data-sent data-recvd dest-addr)
          (*pset :no-collisions data-sent data-recvd dest-addr)
```

```
          (*pset :no-collisions data-sent data-recvd dest-addr)
          (*pset :no-collisions data-sent data-recvd dest-addr))
        :return-statistics-only-p t)
      (vector-push (/ cm-time (* test-loop 10.0)) values)))
    (statistics values time-loop)))))
;;;
;;; Timing #29
;;; 7/24/88
;;; Written by:        David Myers
;;; Description:  This program tests the time required for PEs to get information
;;;           from their eight nearest neighbors. The NEWS operation is
;;;           is used, and values received are stored. Border PEs are
;;;           not members of the active set. All numbers used in the
;;;           grid command are broadcast prior to their use.
;;; Active Processors:     All
;;; Size of Data Used:     32 bit unisigned integers
;;;
(*defun time-29 (time-loop test-loop)
 (format *fp* "~%Timing 29: NEWS grid - 8 nearest neighbors, numbers not brdcst~%")
 (format *data* "~%Timing 29: NEWS grid - 8 nearest neighbors, nmbrs not brdcst~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((x-addr (!! (the fixnum 0)))
       (y-addr (!! (the fixnum 0)))
       (zero (!! (the fixnum 0)))
       (one (!! (the fixnum 1)))
       (neg-one (!! (the fixnum -1)))
       (a (random!! (!! (the fixnum (ash 1 32)))))
       (data-recvd (random!! (!! (the fixnum (ash 1 32))))))
    (declare (type (field-pvar cm:*physical-x-dimension-limit*) x-addr))
    (declare (type (field-pvar cm:*physical-y-dimension-limit*) y-addr))
    (declare (type (field-pvar 32) a))
    (declare (type (field-pvar 2) zero))
    (declare (type (field-pvar 2) one))
    (declare (type (signed-pvar 2) neg-one))
    (declare (type (field-pvar 32) data-recvd))
    (with-paris-from-*lisp
     (cm:my-x-address (pvar-location x-addr))
     (cm:my-y-address (pvar-location y-addr)))
    (*when (and!! (/=!! x-addr (!! (the fixnum 0))
               (!! (the fixnum (- cm:*physical-x-dimension-limit* 1))))
          (/=!! y-addr (!! (the fixnum 0))
               (!! (the fixnum (- cm:*physical-y-dimension-limit* 1)))))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
         (*set data-recvd (pref-grid-relative!! a neg-one zero))
         (*set data-recvd  (pref-grid-relative!! a one zero))
         (*set data-recvd  (pref-grid-relative!! a zero neg-one))
         (*set data-recvd  (pref-grid-relative!! a zero one))
         (*set data-recvd  (pref-grid-relative!! a one one))
         (*set data-recvd  (pref-grid-relative!! a one neg-one))
         (*set data-recvd  (pref-grid-relative!! a neg-one neg-one))
         (*set data-recvd  (pref-grid-relative!! a neg-one one)))
        :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (statistics values time-loop))))))
;;;
;;; Timing #30
;;; 7/24/88
;;; Written by:        David Myers
;;; Description:  This program reports the time it takes the CM-2 to
;;;           pass data in ring.
;;; Active Processors:     All
;;; Size of Data Used:     32 bit unisigned integers
;;;
(*defun time-30 (time-loop test-loop)
 (format *fp* "~%Timing 30: ring send~%")
 (format *data* "~%Timing 30: ring send~%")
 (let ((values 0))
```

```
(setq values (make-array '(100) :fill-pointer 0))
(*all
 (*let ((dest-addr (!! (the fixnum 0)))
       (gray-dest-addr (!! (the fixnum 0)))
       (data-sent (the (pvar (unsigned-byte cm:*cube-address-length*))
               (self-address!!)))
       (data-recvd (!! (the fixnum 0))))
   (declare (type (field-pvar cm:*cube-address-length*) dest-addr))
   (declare (type (field-pvar cm:*cube-address-length*) gray-dest-addr))
   (declare (type (field-pvar 32) data-sent))
   (declare (type (field-pvar 32) data-recvd))
   (*set dest-addr (mod!! (+!! (!! (the fixnum 1))
       (the (pvar (unsigned-byte cm:*cube-address-length*))(self-address!!)))
       (!! (the fixnum cm:*user-cube-address-limit*))))
   (with-paris-from-*lisp
    (cm:gray-code-from-integer (pvar-location gray-dest-addr)
          (pvar-location dest-addr) cm:*cube-address-length*))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (*pset :no-collisions data-sent data-recvd dest-addr))
     :return-statistics-only-p t)
    (vector-push (/ cm-time test-loop) values)))
   (statistics values time-loop)))))
;;;
;;; Timing #31
;;; 7/24/88
;;; Written by:           David Myers
;;; Description:   This program reports the time it takes the CM-2 to
;;;             pass data in a linear array.
;;; Active Processors:     All
;;; Size of Data Used:     32 bit unisigned integers
;;;
(*defun time-31 (time-loop test-loop)
 (format *fp* "~%Timing 31: ring send - max PE number not active~%")
 (format *data* "~%Timing 31: ring send - max PE number not active~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((dest-addr (!! (the fixnum 0)))
         (gray-dest-addr (!! (the fixnum 0)))
         (data-sent (the (pvar (unsigned-byte cm:*cube-address-length*))
                 (self-address!!)))
         (data-recvd (!! (the fixnum 0))))
     (declare (type (field-pvar cm:*cube-address-length*) dest-addr))
     (declare (type (field-pvar cm:*cube-address-length*) gray-dest-addr))
     (declare (type (field-pvar 32) data-sent))
     (declare (type (field-pvar 32) data-recvd))
     (*set dest-addr (mod!! (+!! (!! (the fixnum 1))
         (the (pvar (unsigned-byte cm:*cube-address-length*))(self-address!!)))
         (!! (the fixnum cm:*user-cube-address-limit*))))
     (with-paris-from-*lisp
      (cm:gray-code-from-integer (pvar-location gray-dest-addr)
            (pvar-location dest-addr) cm:*cube-address-length*))
     (*when (/=!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
         (the (pvar (unsigned-byte cm:*cube-address-length*))(self-address!!)))
      (dotimes (i time-loop)
       (format t "~d~%" i)
       (multiple-value-bind (a cm-time b c)
        (cm:time
         (dotimes (j test-loop)
          (*pset :no-collisions data-sent data-recvd dest-addr))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
      (statistics values time-loop))))))
;;;
;;; Timing #40 - Hamming distance test.
;;; 8/2/88
;;; Written by:           David Myers
;;; Description:  This program reports the amount of time required to complete
;;;            send operations to unique address which are a specified
```

```
;;;             hamming distance from the originating processors.  This test
;;;             is the same as test #12 in hamming-test-off-chip.lisp
;;;             except the send addresses are calculated from MSB to LSB.
;;; Active Processors:          All
;;; Size of Data Used:          80 bit unsigned integers
;;;
(*defun time-40 (time-loop test-loop)
 (format *fp* "~%time-40: 80 bit reverse hamming test~%")
 (format *data* "~%time-40: 80 bit reverse hamming test~%")
 (*all
  (let ((mask 0)
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 80)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 80) data-value-sent))
      (declare (type (field-pvar 80) data-value-rcvd))
      (do ((k (- cm:*cube-address-length* 1)(- k 1))(l 1 (+ l 1)))((= k -1))
       (format *fp* "Hamming Distance ~D~%" l)
       (setq mask (logior (ash 1 k) mask))
       (*set dest-address (logxor!! (!! (the fixnum mask))
               (the (field-pvar cm:*cube-address-length*)(self-address!!))))
       (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
         (cm:time
          (dotimes (j test-loop)
           (*pset :no-collisions data-value-sent data-value-rcvd
                                          dest-address)
           (*pset :no-collisions data-value-sent data-value-rcvd
                                          dest-address)
           (*pset :no-collisions data-value-sent data-value-rcvd
                                          dest-address)
           (*pset :no-collisions data-value-sent data-value-rcvd
                                          dest-address)
           (*pset :no-collisions data-value-sent data-value-rcvd
                                          dest-address)
           (*pset :no-collisions data-value-sent data-value-rcvd
                                          dest-address)
           (*pset :no-collisions data-value-sent data-value-rcvd
                                          dest-address)
           (*pset :no-collisions data-value-sent data-value-rcvd
                                          dest-address)
           (*pset :no-collisions data-value-sent data-value-rcvd
                                          dest-address)
           (*pset :no-collisions data-value-sent data-value-rcvd
                                          dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time (* test-loop 10.0)) values)))
       (concentration dest-address)
       (statistics values time-loop))))))
;;;
;;; Timing #42 - Hamming distance test.
;;; 8/2/88
;;; Written by:             David Myers
;;; Description:  This program reports the amount of time required to complete
;;;             send operations when the number of active PEs/chip
;;;             is varied.  All active PEs are sending to PEs located
;;;             at other chips.
;;; Active Processors:          All
;;; Size of Data Used:          80 bit unsigned integers
;;;
(*defun time-42 (time-loop test-loop)
 (format *fp* "~%time-42: # active pes-per-chip~%")
 (format *data* "~%time-42: # active pes-per-chip~%")
 (*all
  (let ((mask 8191)
                (l-index (/ test-loop 10))
        (values 0))
    (setq values (make-array '(50) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
```

```
    (data-value-sent (random!! (!! (the fixnum (ash 1 80)))))
    (on (!! (the fixnum 0)))
    (data-value-rcvd (!! (the fixnum 0))))
(declare (type (field-pvar cm:*cube-address-length*) dest-address))
(declare (type (field-pvar 2) on))
(declare (type (field-pvar 80) data-value-sent))
(declare (type (field-pvar 80) data-value-rcvd))
(*set dest-address (logxor!! (!! (the fixnum mask))
        (the (field-pvar cm:*cube-address-length*)(self-address!!))))
(dotimes (k 16)
 (format *fp* "# of PEs ~D~%" (+ k 1))
 (dotimes (l (+ k 1))
  (*when (=!! (mod!! (the (field-pvar cm:*cube-address-length*)
          (self-address!!))(!! (the fixnum 16)))(!! (the fixnum l)))
   (*set on (!! (the fixnum 1)))))
 (*when (=!! on (!! (the fixnum 1)))
  (dotimes (i time-loop)
   (format t "~d~%" i)
   (multiple-value-bind (a cm-time b c)
    (cm:time
     (dotimes (j l-index)
      (*pset :no-collisions data-value-sent data-value-rcvd
                                    dest-address)
      (*pset :no-collisions data-value-sent data-value-rcvd
                                    dest-address)
      (*pset :no-collisions data-value-sent data-value-rcvd
                                    dest-address)
      (*pset :no-collisions data-value-sent data-value-rcvd
                                    dest-address)
      (*pset :no-collisions data-value-sent data-value-rcvd
                                    dest-address)
      (*pset :no-collisions data-value-sent data-value-rcvd
                                    dest-address)
      (*pset :no-collisions data-value-sent data-value-rcvd
                                    dest-address)
      (*pset :no-collisions data-value-sent data-value-rcvd
                                    dest-address)
      (*pset :no-collisions data-value-sent data-value-rcvd
                                    dest-address)
      (*pset :no-collisions data-value-sent data-value-rcvd
                                    dest-address))
   :return-statistics-only-p t)
   (vector-push (/ cm-time test-loop) values)))
  (concentration dest-address)
  (statistics values time-loop))
  (*set on (!! (the fixnum 0)))))))
```

```
;;;
;;; Timing #5 - Hamming distance test.
;;; 7/13/88
;;; Written by:          David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:     All
;;; Size of Data Used:     1 bit unsigned integers
;;;
(*defun time-05 (time-loop test-loop)
 (format *fp* "~%time-05: 1 bit hamming test~%")
 (format *data* "~%time-05: 1 bit hamming test~%")
 (*all
  (let ((mask (1- cm:*user-cube-address-limit*))
        (values 0))
   (setq values (make-array '(100) :fill-pointer 0))
   (*let ((dest-address (!! (the fixnum 0)))
          (data-value-sent (!! (the fixnum 1)))
          (data-value-rcvd (!! (the fixnum 0))))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (declare (type (field-pvar 1) data-value-sent))
    (declare (type (field-pvar 1) data-value-rcvd))
    (*set dest-address (logxor!! (!! (the fixnum mask))
          (the (field-pvar cm:*cube-address-length*)(self-address!!))))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :no-collisions data-value-sent data-value-rcvd
                                              dest-address))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop)))))
;;;
;;; Timing #6 - Hamming distance test.
;;; 7/13/88
;;; Written by:          David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:     All
;;; Size of Data Used:     2 bit unsigned integers
;;;
(*defun time-06 (time-loop test-loop)
 (format *fp* "~%time-06: 2 bit hamming test~%")
 (format *data* "~%time-06: 2 bit hamming test~%")
 (*all
  (let ((mask (1- cm:*user-cube-address-limit*))
        (values 0))
   (setq values (make-array '(100) :fill-pointer 0))
   (*let ((dest-address (!! (the fixnum 0)))
          (data-value-sent (!! (the fixnum 1)))
          (data-value-rcvd (!! (the fixnum 0))))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (declare (type (field-pvar 2) data-value-sent))
    (declare (type (field-pvar 2) data-value-rcvd))
    (*set dest-address (logxor!! (!! (the fixnum mask))
          (the (field-pvar cm:*cube-address-length*)(self-address!!))))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :no-collisions data-value-sent data-value-rcvd
                                              dest-address))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop)))))
;;;
```

```
;;; Timing #7 - Hamming distance test.
;;; 7/13/88
;;; Written by:          David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:     All
;;; Size of Data Used:     4 bit unsigned integers
;;;
(*defun time-07 (time-loop test-loop)
 (format *fp* "~%time-07: 4 bit hamming test~%")
 (format *data* "~%time-07: 4 bit hamming test~%")
 (*all
  (let ((mask (1- cm:*user-cube-address-limit*))
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (!! (the fixnum 3)))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 4) data-value-sent))
      (declare (type (field-pvar 4) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
              (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                              dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #8 - Hamming distance test.
;;; 7/13/88
;;; Written by:          David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:     All
;;; Size of Data Used:     8 bit unsigned integers
;;;
(*defun time-08 (time-loop test-loop)
 (format *fp* "~%time-08: 8 bit hamming test~%")
 (format *data* "~%time-08: 8 bit hamming test~%")
 (*all
  (let ((mask (1- cm:*user-cube-address-limit*))
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 8)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 8) data-value-sent))
      (declare (type (field-pvar 8) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
              (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                              dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #9 - Hamming distance test.
```

```
;;; 7/13/88
;;; Written by:              David Myers
;;; Description:  This program reports the amount of time required to complete
;;;             send operations to unique address which are a specified
;;;             hamming distance from the originating processors.
;;; Active Processors:        All
;;; Size of Data Used:        16 bit unsigned integers
;;;
(*defun time-09 (time-loop test-loop)
 (format *fp* "~%time-09: 16 bit hamming test~%")
 (format *data* "~%time-09: 16 bit hamming test~%")
 (*all
  (let ((mask (1- cm:*user-cube-address-limit*))
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 16)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 16) data-value-sent))
      (declare (type (field-pvar 16) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
              (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                              dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #10 - Hamming distance test.
;;; 7/13/88
;;; Written by:              David Myers
;;; Description:  This program reports the amount of time required to complete
;;;             send operations to unique address which are a specified
;;;             hamming distance from the originating processors.
;;; Active Processors:        All
;;; Size of Data Used:        32 bit unsigned integers
;;;
(*defun time-10 (time-loop test-loop)
 (format *fp* "~%time-10: 32 bit hamming test~%")
 (format *data* "~%time-10: 32 bit hamming test~%")
 (*all
  (let ((mask (1- cm:*user-cube-address-limit*))
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 32)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 32) data-value-sent))
      (declare (type (field-pvar 32) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
              (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                              dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #11 - Hamming distance test.
;;; 7/13/88
```

```
;;; Written by:           David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:        All
;;; Size of Data Used:        64 bit unsigned integers
;;;
(*defun time-11 (time-loop test-loop)
 (format *fp* "~%time-11: 64 bit hamming test~%")
 (format *data* "~%time-11: 64 bit hamming test~%")
 (*all
  (let ((mask (1- cm:*user-cube-address-limit*))
        (values 0))
   (setq values (make-array '(100) :fill-pointer 0))
   (*let ((dest-address (!! (the fixnum 0)))
          (data-value-sent (random!! (!! (the fixnum (ash 1 64)))))
          (data-value-rcvd (!! (the fixnum 0))))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (declare (type (field-pvar 64) data-value-sent))
    (declare (type (field-pvar 64) data-value-rcvd))
    (*set dest-address (logxor!! (!! (the fixnum mask))
           (the (field-pvar cm:*cube-address-length*)(self-address!!))))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :no-collisions data-value-sent data-value-rcvd
                                          dest-address))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop)))))
;;;
;;; Timing #12 - Hamming distance test.
;;; 7/13/88
;;; Written by:           David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:        All
;;; Size of Data Used:        80 bit unsigned integers
;;;
(*defun time-12 (time-loop test-loop)
 (format *fp* "~%time-12: 80 bit hamming test~%")
 (format *data* "~%time-12: 80 bit hamming test~%")
 (*all
  (let ((mask 15)
        (values 0))
   (setq values (make-array '(100) :fill-pointer 0))
   (*let ((dest-address (!! (the fixnum 0)))
          (data-value-sent (random!! (!! (the fixnum (ash 1 80)))))
          (data-value-rcvd (!! (the fixnum 0))))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (declare (type (field-pvar 80) data-value-sent))
    (declare (type (field-pvar 80) data-value-rcvd))
    (do ((k  4 (+ k 1)))((= k cm:*cube-address-length*))
     (format *fp* "Hamming Distance ~D~%" (+ 1 k))
     (setq mask (logior (ash 1 k)))
     (*set dest-address (logxor!! (!! (the fixnum mask))
            (the (field-pvar cm:*cube-address-length*)(self-address!!))))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
         (*pset :no-collisions data-value-sent data-value-rcvd
                                           dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
     (concentration dest-address)
     (statistics values time-loop))))))
;;;
```

```
;;; Timing #13 - Hamming distance test.
;;; 7/13/88
;;; Written by:            David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:         All
;;; Size of Data Used:         128 bit unsigned integers
;;;
(*defun time-13 (time-loop test-loop)
 (format *fp* "~%time-13: 128 bit hamming test~%")
 (format *data* "~%time-13: 128 bit hamming test~%")
 (*all
  (let ((mask (1- cm:*user-cube-address-limit*))
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 128)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 128) data-value-sent))
      (declare (type (field-pvar 128) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
            (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                              dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #14 - Hamming distance test.
;;; 7/13/88
;;; Written by:            David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:         All
;;; Size of Data Used:         256 bit unsigned integers
;;;
(*defun time-14 (time-loop test-loop)
 (format *fp* "~%time-14: 256 bit hamming test")
 (format *data* "~%time-14: 256 bit hamming test")
 (*all
  (let ((mask (1- cm:*user-cube-address-limit*))
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 256)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 256) data-value-sent))
      (declare (type (field-pvar 256) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
            (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                              dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
```

```
;;;
;;; Timing #5 - Hamming distance test.
;;; 7/13/88
;;; Written by:            David Myers
;;; Description:  This program reports the amount of time required to complete
;;;             send operations to unique address which are a specified
;;;             hamming distance from the originating processors.
;;; Active Processors:       All
;;; Size of Data Used:       1 bit unsigned integers
;;;
(*defun time-05 (time-loop test-loop)
 (format *fp* "~%time-05: 1 bit hamming test~%" )
 (format *data* "~%time-05: 1 bit hamming test~%" )
 (*all
  (let ((mask 15)
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (!! (the fixnum 1)))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 1) data-value-sent))
      (declare (type (field-pvar 1) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
            (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                                dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #6 - Hamming distance test.
;;; 7/13/88
;;; Written by:            David Myers
;;; Description:  This program reports the amount of time required to complete
;;;             send operations to unique address which are a specified
;;;             hamming distance from the originating processors.
;;; Active Processors:       All
;;; Size of Data Used:       2 bit unsigned integers
;;;
(*defun time-06 (time-loop test-loop)
 (format *fp* "~%time-06: 2 bit hamming test~%" )
 (format *data* "~%time-06: 2 bit hamming test~%" )
 (*all
  (let ((mask 15)
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (!! (the fixnum 1)))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 2) data-value-sent))
      (declare (type (field-pvar 2) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
            (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                                dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
```

```
;;; Timing #7 - Hamming distance test.
;;; 7/13/88
;;; Written by:             David Myers
;;; Description:  This program reports the amount of time required to complete
;;;              send operations to unique address which are a specified
;;;              hamming distance from the originating processors.
;;; Active Processors:      All
;;; Size of Data Used:      4 bit unsigned integers
;;;
(*defun time-07 (time-loop test-loop)
  (format *fp* "~%time-07: 4 bit hamming test~%" )
  (format *data* "~%time-07: 4 bit hamming test~%" )
  (*all
   (let ((mask 15)
         (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (!! (the fixnum 3)))
           (data-value-rcvd (!! (the fixnum 0))))
     (declare (type (field-pvar cm:*cube-address-length*) dest-address))
     (declare (type (field-pvar 4) data-value-sent))
     (declare (type (field-pvar 4) data-value-rcvd))
     (*set dest-address (logxor!! (!! (the fixnum mask))
           (the (field-pvar cm:*cube-address-length*)(self-address!!))))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
         (*pset :no-collisions data-value-sent data-value-rcvd
                                         dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
     (concentration dest-address)
     (statistics values time-loop)))))
;;;
;;; Timing #8 - Hamming distance test.
;;; 7/13/88
;;; Written by:             David Myers
;;; Description:  This program reports the amount of time required to complete
;;;              send operations to unique address which are a specified
;;;              hamming distance from the originating processors.
;;; Active Processors:      All
;;; Size of Data Used:      8 bit unsigned integers
;;;
(*defun time-08 (time-loop test-loop)
  (format *fp* "~%time-08: 8 bit hamming test~%" )
  (format *data* "~%time-08: 8 bit hamming test~%" )
  (*all
   (let ((mask 15)
         (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 8)))))
           (data-value-rcvd (!! (the fixnum 0))))
     (declare (type (field-pvar cm:*cube-address-length*) dest-address))
     (declare (type (field-pvar 8) data-value-sent))
     (declare (type (field-pvar 8) data-value-rcvd))
     (*set dest-address (logxor!! (!! (the fixnum mask))
           (the (field-pvar cm:*cube-address-length*)(self-address!!))))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
         (*pset :no-collisions data-value-sent data-value-rcvd
                                         dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
     (concentration dest-address)
     (statistics values time-loop)))))
;;;
;;; Timing #9 - Hamming distance test.
```

```
;;; 7/13/88
;;; Written by:          David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:       All
;;; Size of Data Used:       16 bit unsigned integers
;;;
(*defun time-09 (time-loop test-loop)
 (format *fp* "~%time-09: 16 bit hamming test~%" )
 (format *data* "~%time-09: 16 bit hamming test~%" )
 (*all
  (let ((mask 15)
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 16)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 16) data-value-sent))
      (declare (type (field-pvar 16) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
              (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                              dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #10 - Hamming distance test.
;;; 7/13/88
;;; Written by:          David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:       All
;;; Size of Data Used:       32 bit unsigned integers
;;;
(*defun time-10 (time-loop test-loop)
 (format *fp* "~%time-10: 32 bit hamming test~%" )
 (format *data* "~%time-10: 32 bit hamming test~%" )
 (*all
  (let ((mask 15)
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 32)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 32) data-value-sent))
      (declare (type (field-pvar 32) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
              (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                              dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #11 - Hamming distance test.
;;; 7/13/88
```

```
;;; Written by:              David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:        All
;;; Size of Data Used:        64 bit unsigned integers
;;;
(*defun time-11 (time-loop test-loop)
 (format *fp* "~%time-11: 64 bit hamming test~%" )
 (format *data* "~%time-11: 64 bit hamming test~%")
 (*all
  (let ((mask 15)
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 64)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 64) data-value-sent))
      (declare (type (field-pvar 64) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
              (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                                  dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #12 - Hamming distance test.
;;; 7/13/88
;;; Written by:              David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:        All
;;; Size of Data Used:        80 bit unsigned integers
;;;
(*defun time-12 (time-loop test-loop)
 (format *fp* "~%time-12: 80 bit hamming test~%")
 (format *data* "~%time-12: 80 bit hamming test~%")
 (*all
  (let ((mask 0)
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 80)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 80) data-value-sent))
      (declare (type (field-pvar 80) data-value-rcvd))
      (dotimes (k 4)
        (format *fp* "Hamming Distance ~D~%" (+ 1 k))
        (setq mask (logior (ash 1 k)))
        (*set dest-address (logxor!! (!! (the fixnum mask))
                (the (field-pvar cm:*cube-address-length*)(self-address!!))))
        (dotimes (i time-loop)
          (format t "~d~%" i)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (dotimes (j test-loop)
                (*pset :no-collisions data-value-sent data-value-rcvd
                                                    dest-address))
            :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values)))
        (concentration dest-address)
        (statistics values time-loop))))))
;;;
```

```
;;; Timing #13 - Hamming distance test.
;;; 7/13/88
;;; Written by:            David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:        All
;;; Size of Data Used:        128 bit unsigned integers
;;;
(*defun time-13 (time-loop test-loop)
 (format *fp* "~%time-13: 128 bit hamming test~%")
 (format *data* "~%time-13: 128 bit hamming test~%")
 (*all
  (let ((mask 15)
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 128)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 128) data-value-sent))
      (declare (type (field-pvar 128) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
              (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                                   dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #14 - Hamming distance test.
;;; 7/13/88
;;; Written by:            David Myers
;;; Description:  This program reports the amount of time required to complete
;;;               send operations to unique address which are a specified
;;;               hamming distance from the originating processors.
;;; Active Processors:        All
;;; Size of Data Used:        256 bit unsigned integers
;;;
(*defun time-14 (time-loop test-loop)
 (format *fp* "~%time-14: 256 bit hamming test~%")
 (format *data* "~%time-14: 256 bit hamming test~%")
 (*all
  (let ((mask 15)
        (values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*let ((dest-address (!! (the fixnum 0)))
           (data-value-sent (random!! (!! (the fixnum (ash 1 256)))))
           (data-value-rcvd (!! (the fixnum 0))))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (declare (type (field-pvar 256) data-value-sent))
      (declare (type (field-pvar 256) data-value-rcvd))
      (*set dest-address (logxor!! (!! (the fixnum mask))
              (the (field-pvar cm:*cube-address-length*)(self-address!!))))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :no-collisions data-value-sent data-value-rcvd
                                                   dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
```

```
;;;
;;; Timing #32
;;; 7/26/88
;;; Written by:          David Myers
;;; Description:  This program tests the time required for an unsigned add.
;;; Active Processors:     All
;;; Size of Data Used:     32 bit unisigned integers
;;;
(*defun time-32 (time-loop test-loop)
 (format *fp* "~%Timing 32: unsigned add~%")
 (format *data* "~%Timing 32: unsigned add~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((x (!! (the fixnum 100999)))
          (y (!! (the fixnum 372826)))
          (dest (!! (the fixnum 0))))
     (declare (type (field-pvar 32) x))
     (declare (type (field-pvar 32) y))
     (declare (type (field-pvar 32) dest))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
        (with-paris-from-*lisp
         (cm:unsigned-add (pvar-location dest)(pvar-location x)
             (pvar-location y)(pvar-length dest)(pvar-length x)
             (pvar-length y))))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
     (statistics values time-loop))))))
;;;
;;; Timing #33
;;; 7/26/88
;;; Written by:          David Myers
;;; Description:  This program tests the time required for an unsigned multiply.
;;; Active Processors:     All
;;; Size of Data Used:     33 bit unisigned integers
;;;
(*defun time-33 (time-loop test-loop)
 (format *fp* "~%Timing 33: unsigned multiply~%")
 (format *data* "~%Timing 33: unsigned multiply~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((x (!! (the fixnum 3874738)))
          (y (!! (the fixnum 2743878)))
          (dest (!! (the fixnum 0))))
     (declare (type (field-pvar 32) x))
     (declare (type (field-pvar 32) y))
     (declare (type (field-pvar 32) dest))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
        (with-paris-from-*lisp
         (cm:unsigned-multiply (pvar-location dest)(pvar-location x)
              (pvar-location y)(pvar-length dest)(pvar-length x)
              (pvar-length y))))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
     (statistics values time-loop)))))
;;;
;;; Timing #34
;;; 7/26/88
;;; Written by:          David Myers
;;; Description:  This program tests the time required for an unsigned subtract.
;;; Active Processors:     All
;;; Size of Data Used:     34 bit unisigned integers
;;;
(*defun time-34 (time-loop test-loop)
```

```
 (format *fp* "~%Timing 34: unsigned subtract~%")
 (format *data* "~%Timing 34: unsigned subtract~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((x (!! (the fixnum 7645838)))
          (y (!! (the fixnum 3838629)))
          (dest (!! (the fixnum 0))))
     (declare (type (field-pvar 32) x))
     (declare (type (field-pvar 32) y))
     (declare (type (field-pvar 32) dest))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
        (with-paris-from-*lisp
         (cm:unsigned-subtract (pvar-location dest)(pvar-location x)
              (pvar-location y)(pvar-length dest)(pvar-length x)
              (pvar-length y))))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
     (statistics values time-loop)))))
;;;
;;; Timing #35
;;; 7/26/88
;;; Written by:          David Myers
;;; Description:  This program tests the time required for an unsigned divide.
;;; Active Processors:    All
;;; Size of Data Used:    32 bit unisigned integers
;;;
(*defun time-35 (time-loop test-loop)
 (format *fp* "~%Timing 35: unsigned divide~%")
 (format *data* "~%Timing 35: unsigned divide~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((x (!! (the fixnum 5892633)))
          (y (!! (the fixnum 9752466)))
          (dest (!! (the fixnum 0))))
     (declare (type (field-pvar 32) x))
     (declare (type (field-pvar 32) y))
     (declare (type (field-pvar 32) dest))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
        (with-paris-from-*lisp
         (cm:unsigned-truncate-divide (pvar-location dest)(pvar-location x)
              (pvar-location y)(pvar-length dest)(pvar-length x)
              (pvar-length y))))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
     (statistics values time-loop)))))
;;;
;;; Timing #36
;;; 7/26/88
;;; Written by:          David Myers
;;; Description:  This program tests the time required for a floating-point add.
;;; Active Processors:    All
;;; Size of Data Used:    32 bit unisigned integers
;;;
(*defun time-36 (time-loop test-loop)
 (format *fp* "~%Timing 36: floating point add~%")
 (format *data* "~%Timing 36: floating point add~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((x (!! (the single-float 0.2)))
          (y (!! (the single-float 1.1)))
          (dest (!! (the single-float 0.0))))
     (declare (type (pvar single-float) x))
```

```
        (declare (type (pvar single-float) y))
        (declare (type (pvar single-float) dest))
        (dotimes (i time-loop)
          (format t "~d~%" i)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (dotimes (j test-loop)
              (with-paris-from-*lisp
                (cm:f+ (pvar-location x)(pvar-location y))))
              :return-statistics-only-p t)
              (*set x (!! (the single-float 0.0)))
            (vector-push (/ cm-time test-loop) values)))
          (statistics values time-loop)))))
;;;
;;; Timing #37
;;; 7/26/88
;;; Written by:          David Myers
;;; Description:  This program tests the time required for a floating-point subt.
;;; Active Processors:      All
;;; Size of Data Used:      32 bit unisigned integers
;;;
(*defun time-37 (time-loop test-loop)
  (format *fp* "~%Timing 37: floating point subtract~%")
  (format *data* "~%Timing 37: floating point subtract~%")
  (let ((values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*all
      (*let ((x (!! (the single-float 100000.0)))
             (y (!! (the single-float 1.0)))
             (dest (!! (the single-float 0.0))))
        (declare (type (pvar single-float) x))
        (declare (type (pvar single-float) y))
        (declare (type (pvar single-float) dest))
        (dotimes (i time-loop)
          (format t "~d~%" i)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (dotimes (j test-loop)
              (with-paris-from-*lisp
                (cm:f- (pvar-location x)(pvar-location y))))
              :return-statistics-only-p t)
              (*set x (!! (the single-float 100000.0)))
            (vector-push (/ cm-time test-loop) values)))
          (statistics values time-loop)))))
;;;
;;; Timing #38
;;; 7/26/88
;;; Written by:          David Myers
;;; Description:  This program tests the time required for a floating-point mult.
;;; Active Processors:      All
;;; Size of Data Used:      32 bit unisigned integers
;;;
(*defun time-38 (time-loop test-loop)
  (format *fp* "~%Timing 38: floating point multiply~%")
  (format *data* "~%Timing 38: floating point multipl~%")
  (let ((values 0)
                    (inner-loop (/ test-loop 10)))
    (setq values (make-array '(100) :fill-pointer 0))
    (*all
      (*let ((x (!! (the single-float 2.0)))
             (y (!! (the single-float 1.72)))
             (dest (!! (the single-float 0.0))))
        (declare (type (pvar single-float) x))
        (declare (type (pvar single-float) y))
        (declare (type (pvar single-float) dest))
        (dotimes (i time-loop)
          (format t "~d~%" i)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (dotimes (j inner-loop)
              (with-paris-from-*lisp
                (cm:f* (pvar-location x)(pvar-location y)))
              (with-paris-from-*lisp
```

```
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y))))
         :return-statistics-only-p t)
         (*set x (!! (the single-float 2.0)))
        (vector-push (/ cm-time test-loop) values)))
       (statistics values time-loop)))))
;;;
;;; Timing #38-2
;;; 7/26/88
;;; Written by:          David Myers
;;; Description:  This program tests the time required for a floating-point mult.
;;; Active Processors:     All
;;; Size of Data Used:     32 bit unisigned integers
;;;
(*defun time-38-2 (time-loop test-loop)
 (format *fp* "~%Timing 38: floating point multiply: using nice numbers~%")
 (format *data* "~%Timing 38: floating point multipl~%")
 (let ((values 0)
               (inner-loop (/ test-loop 10)))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((x (!! (the single-float 2.0)))
          (y (!! (the single-float 1.0)))
          (dest (!! (the single-float 0.0))))
     (declare (type (pvar single-float) x))
     (declare (type (pvar single-float) y))
     (declare (type (pvar single-float) dest))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j inner-loop)
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y)))
         (with-paris-from-*lisp
          (cm:f* (pvar-location x)(pvar-location y))))
         :return-statistics-only-p t)
         (*set x (!! (the single-float 2.0)))
        (vector-push (/ cm-time test-loop) values)))
       (statistics values time-loop)))))
;;;
;;; Timing #39
```

```
;;; 7/26/88
;;; Written by:          David Myers
;;; Description:  This program tests the time required for a floating-point mult.
;;; Active Processors:     All
;;; Size of Data Used:     32 bit unisigned integers
;;;
(*defun time-39 (time-loop test-loop)
  (format *fp* "~%Timing 39: floating point divide~%")
  (format *data* "~%Timing 39: floating point divide~%")
  (let ((values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*all
     (*let ((x (!! (the single-float 45098732.0)))
            (y (!! (the single-float 1.72)))
            (dest (!! (the single-float 0.0))))
       (declare (type (pvar single-float) x))
       (declare (type (pvar single-float) y))
       (declare (type (pvar single-float) dest))
       (dotimes (i time-loop)
         (format t "~d~%" i)
         (multiple-value-bind (a cm-time b c)
           (cm:time
             (dotimes (j test-loop)
             (with-paris-from-*lisp
               (cm:f/ (pvar-location x)(pvar-location y))))
             :return-statistics-only-p t)
             (*set x (!! (the single-float 2.0)))
           (vector-push (/ cm-time test-loop) values)))
         (statistics values time-loop)))))
(defun main ()
  (setq *fp* (open "arith-stats" :direction :output))
  (setq *data* (open "arith-data" :direction :output))
  (time-32 100 20000)
  (time-33 100 5000)
  (time-34 100 50000)
  (time-35 100 4000)
  (time-36 100 8000)
  (time-37 100 8000)
  (time-38 100 8000)
  (time-39 100 4000)
  (close *data*)
  (close *fp*))
```

```
(defvar *fp*)
(defvar *data*)

(defun enable-all ()
 (cm:move-constant-always cm:context-flag 1 1)
 (cm:move-constant-always cm:overflow-flag 0 1))

(*defun concentration (address-pvar)
 (declare (type (field-pvar cm:*cube-address-length*) address-pvar))
 (*let ((rcvd-count))
  (declare (type (field-pvar 16) rcvd-count))
  (*all
   (*set rcvd-count (!! (the fixnum 0))))
  (*pset :add (!! (the fixnum 1)) rcvd-count address-pvar)
  (let ((n-a-p (cm:global-count cm:context-flag)))
   (format *fp* "  number of active processors: ~D~%" n-a-p)
   (*all
    (*when (/=!! rcvd-count (!! (the fixnum 0)))
     (let* ((n-receiving (cm:global-count cm:context-flag))
            (total-sum (*sum rcvd-count))
            (max-rcvd (*max rcvd-count))
            (av-received (/ (float total-sum) (float n-receiving))))
      (format *fp* "  number receiving: ~D (~,1F%)~%" n-receiving
             (* 100.0 (/ n-receiving (float n-a-p))))
      (format *fp* "  average received: ~,2F~%" av-received)
      (format *fp* "  max received: ~D~%" max-rcvd)))))))

(defun statistics (values time-loop)
 (let ((average  0)
      (std-dev  0)
      (sum      0)
      (sqr-sum  0)
      (maximum  0)
      (minimum  0)
      (temp     0))
  (dotimes (i time-loop)
   (setq temp (vector-pop values))
   (format *data* "~F~%" temp)
   (setq sum (+ sum temp))
   (setq sqr-sum (+ sqr-sum (* temp temp)))
   (setq maximum (max maximum temp))
   (if (= i 0)
    (setq minimum temp))
   (setq minimum (min minimum temp)))
  (setq average (/ sum time-loop))
  (setq std-dev (sqrt (- (/ sqr-sum (- time-loop 1))
           (/ (* time-loop (* average average))(- time-loop 1)))))
  (format *fp* "  Average: ~f~%" average)
  (format *fp* "  Standard Deviation: ~f~%" std-dev)
  (format *fp* "  Maximum: ~f~%" maximum)
  (format *fp* "  Minimum: ~f~%" minimum)))
;;;
;;; Timing #4
;;; 6/30/88
;;; Written by:           David Myers
;;; Description: This program times the calculation of an
;;;           inner product on the CM2.
;;; Active Processors:        All
;;; Size of data:          32 bit unsigned integers
;;;
(*defun time-4 (time-loop test-loop)
 (format *fp* "time-4~%")
 (format *data* "time-4~%")
 (let ((values 0))
  (enable-all)
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((a (!! (the fixnum 1)))
        (b (!! (the fixnum 4)))
        (c (!! (the fixnum 0)))
        (result (!! (the fixnum 0))))
    (declare (type (field-pvar 32) a))
    (declare (type (field-pvar 32) b))
```

```
        (declare (type (field-pvar 32) c))
        (declare (type (field-pvar 32) result))
        ;;
        ;; Instruction to be timed.
        ;;
        (dotimes (i time-loop)
          (format t "~d~%" i)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (dotimes (i test-loop)
                (*set c (*!! a b))
                (*set result (scan!! c '+!!)))
            :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values)))
        (statistics values time-loop)))))
;;;
;;; Timing #15
;;; 7/18/88
;;; Written by:         David Myers
;;; Description:  This program reports the time it take the connection
;;;               machine to do the following calculation,
;;;               u(i) = sin( 2*pi*i / n ).
;;;
;;; Active Processors:     All
;;; Size of Data Used:     !!!!!!!!!
;;;
(*defun time-15 (time-loop test-loop)
  (format *fp* "~%time-15: u(i) = sin( 2*pi*i / n ) ~%")
  (format *data* "~%time-15: u(i) = sin( 2*pi*i / n ) ~%")
  (let ((values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*all
      (*let ((i (float!! (+!! (!! (the fixnum 1))(the (field-pvar
                        cm:*cube-address-length*)(self-address!!)))))
            (two-pi (*!! (!! (the single-float 2.0))
                             (!! (the single-float 3.145927))))
            (number-pes (!! (the single-float (float
                             cm:*user-cube-address-limit*))))
            (u (!! (the single-float 0.0))))
        (declare (type (pvar single-float) i))
        (declare (type (pvar single-float) number-pes))
        (declare (type (pvar single-float) u))
        (declare (type (pvar single-float) two-pi))
        (dotimes (k time-loop)
          (format t "~d~%" k)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (dotimes (j test-loop)
                (*set u (sin!! (/!! (*!! two-pi i) number-pes))))
            :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values)))
        (statistics values time-loop)))))
;;;
;;; Timing #19
;;; 7/18/88
;;; Written by:         David Myers
;;; Description:  This program reports the time it take the connection
;;;               machine to calculate a random number in each PE.
;;; Active Processors:     All
;;; Size of Data Used:     32 bit unisigned integers
;;;
(*defun time-19 (time-loop test-loop)
  (format *fp* "~%Timing 19: random!!~%")
  (format *data* "~%Timing 19: random!!~%")
  (let ((values 0))
    (setq values (make-array '(100) :fill-pointer 0))
    (*all
      (*let ((number (!! (the fixnum 0)))
            (seed (!! (the fixnum (ash 1 32)))))
        (declare (type (field-pvar cm:*cube-address-length*) number))
        (declare (type (field-pvar cm:*cube-address-length*) seed))
        (dotimes (i time-loop)
          (format t "~d~%" i)
```

```
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (*set number (random!! seed)))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (statistics values time-loop)))))
;;;
;;; Timing #27
;;; 7/23/88
;;; Written by:         David Myers
;;; Description:  This program reports the time it takes the CM-2 to perform
;;;               image smoothing.  NEWS operations are used.  Float values
;;;               are used instead of integers because the current
;;;               implementation of /!! always returns a float value.
;;; Active Processors:  All
;;; Size of Data Used:  32 bit unisigned integers
;;;
(*defun time-27 (time-loop test-loop)
 (format *fp* "~%Timing 27: NEWS grid - image smoothing~%")
 (format *data* "~%Timing 27: NEWS grid - image smoothing~%")
 (let ((values 0))
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((x-addr (!! (the fixnum 0)))
          (y-addr (!! (the fixnum 0)))
          (sum (!! (the fixnum 0)))
          (a (random!! (!! (the fixnum (ash 1 8)))))
          (data-recvd (random!! (!! (the fixnum (ash 1 8))))))
    (declare (type (field-pvar cm:*physical-x-dimension-limit*) x-addr))
    (declare (type (field-pvar cm:*physical-y-dimension-limit*) y-addr))
    (declare (type (pvar single-float) a))
    (declare (type (pvar single-float) data-recvd))
    (declare (type (pvar single-float) sum))
    (with-paris-from-*lisp
     (cm:my-x-address (pvar-location x-addr))
     (cm:my-y-address (pvar-location y-addr)))
    (*when (and!! (/=!! x-addr (!! (the fixnum 0))
                        (!! (the fixnum (- cm:*physical-x-dimension-limit* 1))))
                  (/=!! y-addr (!! (the fixnum 0))
                        (!! (the fixnum (- cm:*physical-y-dimension-limit* 1)))))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
         (*set sum (/!! (+!! a
           (pref-grid-relative!! a (!! (the fixnum -1))(!! (the fixnum 0)))
           (pref-grid-relative!! a (!! (the fixnum 1))(!! (the fixnum 0)))
           (pref-grid-relative!! a (!! (the fixnum 0))(!! (the fixnum -1)))
           (pref-grid-relative!! a (!! (the fixnum 0))(!! (the fixnum 1)))
           (pref-grid-relative!! a (!! (the fixnum 1))(!! (the fixnum 1)))
           (pref-grid-relative!! a (!! (the fixnum 1))(!! (the fixnum -1)))
           (pref-grid-relative!! a (!! (the fixnum -1))
                                   (!! (the fixnum -1)))
           (pref-grid-relative!! a (!! (the fixnum -1))
                    (!! (the fixnum 1))))(!! (the fixnum 9)))))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
     (statistics values time-loop))))))
(defun main ()
 (setq *fp* (open "algorithm-stats" :direction :output))
 (setq *data* (open "algorithm-data" :direction :output))
 (time-4 100 3000)
 (time-15 100 100)
 (time-19 100 4000)
 (time-27 100 600)
 (close *data*)
 (close *fp*))
```

```
;;;
;;; Timing #1
;;; 7/5/88
;;; Written by:         Bill O'Farrell
;;; Modified by:        Descritpion
;;; Description:  This program reports the time required for multiplication
;;;             of two unsigned integers.
;;; Active Processors:    All
;;; Data Size:          2 bit and 97 bit unsigned integers
;;;
(*defun time-test01 (time-loop test-loop)
 (format *fp* "~%TIME-TEST01~%")
 (format *data* "~%TIME-TEST01~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((three (!! (the fixnum 1)))
        (curr (random!! (!! (the fixnum (ash 1 94))))))
    (declare (type (field-pvar 2) three))
    (declare (type (field-pvar 97) curr))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
       (*set curr (*!! curr three))) :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (statistics values time-loop))))
;;;
;;; Timing #3
;;; 7/5/88
;;; Written by:              Bill O'Farrell
;;; Modified by:             David Myers
;;; Description:  This program performs the same test as time1.lisp
;;;           except  a 35 bit number is used instead of a two bit
;;;           number.
;;; Active PEs:              All
;;; Size of Data:            35 and 97 bit unsigned integers
;;;
(*defun time-test03 (time-loop test-loop)
 (format *fp* "~%TIME-TEST03~%")
 (format *data* "~%TIME-TEST03~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v35 (!!.(the fixnum (ash 1 35))))
        (curr (random!! (!! (the fixnum (ash 1 94))))))
    (declare (type (field-pvar 35) v35))
    (declare (type (field-pvar 97) curr))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
       (*set curr (*!! curr v35))) :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (statistics values time-loop))))
;;;
;;; Timing 4
;;; 7/5/88
;;; Written by:             Bill O'Farrell
;;; Modified by:            David W. Myers
;;; Description:  This program times a send with overwrite of two
;;;           45 bit unsigned integers to random locations
;;; Active Processors:      All
;;; Size of Data Used:      45 bit unsigned integers
;;;
(*defun time-test04 (time-loop test-loop)
 (format *fp* "~%TIME-TEST04~%")
 (format *data* "~%TIME-TEST04~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
```

```
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
        (v45dest (random!! (!! (the fixnum (ash 1 45)))))
        (dest-address (random!! (!! (the fixnum (ash 1
                                   cm:*cube-address-length*))))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
         (*pset :overwrite v45 v45dest dest-address))
              :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop))))
;;;
;;; Timing #5
;;; 7/5/88
;;; Written by:              Bill O'Farrell
;;; Modified by:             David Myers
;;; Description:  Each PE sends to another PE that is a small hamming
;;;               distance away.
;;; Active Processors:         All
;;; Size of Data Used:         45 bit unsigned integers.
;;;
(*defun time-test05 (time-loop test-loop)
 (format *fp* "~%TIME-TEST05~%")
 (format *data* "~%TIME-TEST05~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
        (v45dest (random!! (!! (the fixnum (ash 1 45)))))
        (dest-address (!! (the fixnum 0)))
        (mask (!! (the fixnum 0))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (declare (type (field-pvar cm:*cube-address-length*) mask))
   (dotimes (i 6)
    (*set mask (+!! mask (ash!! (!! (the fixnum 1))
        (random!! (!! (the fixnum cm:*cube-address-length*)))))))
   (*set dest-address (logior!! mask (the (field-pvar
        cm:*cube-address-length*)(self-address!!))))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
         (*pset :overwrite v45 v45dest dest-address))
              :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop))))
;;;
;;; Timing #6
;;; 7/5/88
;;; Written by:              Bill O'Farrell
;;; Modified by:             David Myers
;;; Description:  Each PE sends to another PE that is at most a hamming
;;;               distance of 6 away.
;;; Active Processors:         All
;;; Size of Data Used:         45 bit unsigned integers.
;;;
(*defun time-test06 (time-loop test-loop)
 (format *fp* "~%TIME-TEST06~%")
 (format *data* "~%TIME-TEST06~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
```

```
        (v45dest (random!! (!! (the fixnum (ash 1 45)))))
        (mask (!! (the fixnum 0)))
        (dest-address (!! (the fixnum 0))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (declare (type (field-pvar cm:*cube-address-length*) mask))
    (dotimes (i 6)
     (*set mask (logior!! mask (ash!! (!! (the fixnum 1))
          (random!! (!! (the fixnum cm:*cube-address-length*)))))))
    (*set dest-address (logxor!! mask (the (field-pvar
          cm:*cube-address-length*)(self-address!!))))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :overwrite v45 v45dest dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;;  Timing #7
;;;  7/7/88
;;;  Written by:         Bill O'Farrell
;;;  Modified by:        David W. Myers
;;;  Description:  Send to unique addresses with a Hamming distance of exactly
;;;          9.
;;;  Active Processors:
;;;  Size of Data Used:
;;;
(*defun time-test07 (time-loop test-loop)
 (format *fp* "~%TIME-TEST07~%")
 (format *data* "~%TIME-TEST07~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
             (v45dest (random!! (!! (the fixnum (ash 1 45)))))
             (dest-address)
             (bit-list)
             (found-index)
             (received-something)
             (my-address-taken nil!!)
             (dest-fixed nil!!)
             (return-address))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar 15) dest-address))
    (declare (type (field-pvar 45) bit-list))
    (declare (type (field-pvar 4) found-index))
    (declare (type boolean-pvar received-something))
    (declare (type boolean-pvar my-address-taken))
    (declare (type boolean-pvar dest-fixed))
    (declare (type (field-pvar cm:*cube-address-length*) return-address))
    (do ((its 1 (1+ its)))
             ((<= (*when (not!! dest-fixed) (*sum (!! (the fixnum 1)))) 200))
            (*when (not!! dest-fixed)
             (format t "~D: computing addresses for ~D processors~%" its
                        (*sum (!! (the fixnum 1))))
             (*set dest-address (!! (the fixnum 0)))
             (*set bit-list (!! (the fixnum 0)))
             (*set found-index (!! (the fixnum 0)))
             (do ()
               ((= 0 (*when (<!! found-index (!! (the fixnum 9)))
                        (*sum (!! (the fixnum 1))))))
              (*when (<!! found-index (!! (the fixnum 9)))
               (*let ((ran-position (random!! (!! (the fixnum 15))))
               (not-unique nil!!))
            (declare (type (field-pvar 5) ran-position))
            (declare (type boolean-pvar not-unique))
            (do ((i 0 (1+ i)))
```

```
            ((= i 9))
           (*when (<!! (!! (the fixnum i)) found-index)
            (*when (=!! (load-byte!! bit-list (*!! (!! (the fixnum i))
              (!! (the fixnum 5)))(!! (the fixnum 5)))
          ran-position)
             (*set not-unique t!!))))
          (*when (not!! not-unique)
           (*set bit-list
          (deposit-byte!! bit-list (*!! found-index
                   (!! (the fixnum 5))) (!! (the fixnum 5))
          ran-position))
           (*set found-index (1+!! found-index)))))
           (dotimes (i 9)
            (*set dest-address
           (logior!! dest-address
            (ash!! (!! 1)
            (load-byte!! bit-list (*!! (!! (the fixnum i))
                   (!! (the fixnum 5)))(!! (the fixnum 5)))))))
           (*let ((bit-sum (!! (the fixnum 0)))
          (dest-address-temp dest-address))
            (declare (type (field-pvar 5) bit-sum))
            (declare (type (field-pvar 15) dest-address-temp))
            (dotimes (i 15)
             (*set bit-sum (+!! bit-sum (load-byte!! dest-address-temp
              (!! (the fixnum 0))(!! (the fixnum 1)))))
             (*set dest-address-temp (ash!! dest-address-temp
                          (!! (the fixnum -1)))))
            (*when (/=!! bit-sum (!! (the fixnum 9)))
             (do-for-selected-processors (cube-add)
          (format t "Processor ~D has a bit-sum not equal to 9~%"
                             cube-add))))
           (*set dest-address (logxor!! dest-address
       (the (field-pvar cm:*cube-address-length*)(self-address!!))))
           (*all (*set received-something nil!!))
           (*pset :overwrite
       (the (field-pvar cm:*cube-address-length*)(self-address!!))
       return-address dest-address received-something)
           (*all
            (*when (and!! received-something (not!! my-address-taken))
             (*set my-address-taken t!!)
             (*pset :no-collisions t!! dest-fixed return-address)))))
  (dotimes (i time-loop)
   (format t "~d~%" i)
   (*when dest-fixed
    (format t "~%number active: ~D~%" (*sum (!! (the fixnum 1))))
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (*pset :overwrite v45 v45dest dest-address))
     :return-statistics-only-p t)
    (vector-push (/ cm-time test-loop) values))))
  (concentration dest-address)
  (statistics values time-loop))))
;;;
;;; Timing #8
;;; 7/7/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David W. Myers
;;; Description:  Send to unique addresses with a Hamming distance of exactly
;;;          6.
;;; Active Processors:
;;; Sise of Data Used:
;;;
(*defun time-test08 (time-loop test-loop)
 (format *fp* "~%TIME-TEST08~%")
 (format *data* "~%TIME-TEST08~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (dest-address)
         (bit-list)
```

```
                (found-index)
                (received-something)
                (my-address-taken nil!!)
                (dest-fixed nil!!)
                (return-address))
(declare (type (field-pvar 45) v45))
(declare (type (field-pvar 45) v45dest))
(declare (type (field-pvar 15) dest-address))
(declare (type (field-pvar 45) bit-list))
(declare (type (field-pvar 4) found-index))
(declare (type boolean-pvar received-something))
(declare (type boolean-pvar my-address-taken))
(declare (type boolean-pvar dest-fixed))
(declare (type (field-pvar cm:*cube-address-length*) return-address))
(do ((its 1 (1+ its)))
                ((<= (*when (not!! dest-fixed) (*sum (!! (the fixnum 1)))) 200))
            (*when (not!! dest-fixed)
              (format t "~D: computing addresses for ~D processors~%" its
                                      (*sum (!! (the fixnum 1))))
              (*set dest-address (!! (the fixnum 0)))
              (*set bit-list (!! (the fixnum 0)))
              (*set found-index (!! (the fixnum 0)))
              (do ()
                 ((= 0 (*when (<!! found-index (!! (the fixnum 6)))
                                      (*sum (!! (the fixnum 1))))))
                (*when (<!! found-index (!! (the fixnum 6)))
                  (*let ((ran-position (random!! (!! (the fixnum 15))))
                  (not-unique nil!!))
            (declare (type (field-pvar 5) ran-position))
            (declare (type boolean-pvar not-unique))
            (do ((i 0 (1+ i)))
                ((= i 6))
              (*when (<!! (!! (the fixnum i)) found-index)
                (*when (=!! (load-byte!! bit-list (*!! (!! (the fixnum i))
                   (!! (the fixnum 5)))(!! (the fixnum 5)))
            ran-position)
                  (*set not-unique t!!))))
            (*when (not!! not-unique)
              (*set bit-list
            (deposit-byte!! bit-list (*!! found-index
                        (!! (the fixnum 5))) (!! (the fixnum 5))
            ran-position))
              (*set found-index (1+!! found-index))))))
              (dotimes (i 6)
                (*set dest-address
              (logior!! dest-address
                (ash!! (!! 1)
               (load-byte!! bit-list (*!! (!! (the fixnum i))
                        (!! (the fixnum 5)))(!! (the fixnum 5)))))))
              (*let ((bit-sum (!! (the fixnum 0)))
             (dest-address-temp dest-address))
                (declare (type (field-pvar 5) bit-sum))
                (declare (type (field-pvar 15) dest-address-temp))
                (dotimes (i 15)
                  (*set bit-sum (+!! bit-sum (load-byte!! dest-address-temp
                   (!! (the fixnum 0))(!! (the fixnum 1)))))
                  (*set dest-address-temp (ash!! dest-address-temp
                                     (!! (the fixnum -1)))))
                (*when (/=!! bit-sum (!! (the fixnum 6)))
                  (do-for-selected-processors (cube-add)
            (format t "Processor ~D has a bit-sum not equal to 6~%"
                                          cube-add))))
               (*set dest-address (logxor!! dest-address
         (the (field-pvar cm:*cube-address-length*)(self-address!!))))
               (*all (*set received-something nil!!))
               (*pset :overwrite
         (the (field-pvar cm:*cube-address-length*)(self-address!!))
         return-address dest-address received-something)
               (*all
                 (*when (and!! received-something (not!! my-address-taken))
                   (*set my-address-taken t!!)
                   (*pset :no-collisions t!! dest-fixed return-address)))))
(dotimes (i time-loop)
```

```
      (format t "~d~%" i)
      (*when dest-fixed
        (format t "~%number active: ~D~%" (*sum (!! (the fixnum 1))))
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :overwrite v45 v45dest dest-address))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values))))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;; Timing #11
;;; 7/5/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David Myers
;;; Description:  Each PE sends to another PE that is at most a hamming
;;;               distance of 3 away.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers.
;;;
(*defun time-test11 (time-loop test-loop)
  (format *fp* "~%TIME-TEST11~%")
  (format *data* "~%TIME-TEST11~%")
  (enable-all)
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
          (v45dest (random!! (!! (the fixnum (ash 1 45)))))
          (mask  (!! (the fixnum 0)))
          (dest-address (!! (the fixnum 0))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (declare (type (field-pvar cm:*cube-address-length*) mask))
    (dotimes (i 3)
      (*set mask (logior!! mask (ash!! (!! (the fixnum 1))
          (random!! (!! (the fixnum cm:*cube-address-length*)))))))
    (*set dest-address (logxor!! mask (the (field-pvar
          cm:*cube-address-length*)(self-address!!))))
    (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
        (cm:time
          (dotimes (j test-loop)
            (*pset :overwrite v45 v45dest dest-address))
        :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;; Timing #12
;;; 7/7/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David W. Myers
;;; Description:  Send to unique addresses with a Hamming distance of exactly
;;;               6.
;;; Active Processors:
;;; Size of Data Used:
;;;
(*defun time-test12 (time-loop test-loop)
  (format *fp* "~%TIME-TEST12~%")
  (format *data* "~%TIME-TEST12~%")
  (enable-all)
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
                (v45dest (random!! (!! (the fixnum (ash 1 45)))))
                (dest-address)
                (bit-list)
                (found-index)
                (received-something)
                (my-address-taken nil!!)
```

```
               (dest-fixed nil!!)
               (return-address))
(declare (type (field-pvar 45) v45))
(declare (type (field-pvar 45) v45dest))
(declare (type (field-pvar 15) dest-address))
(declare (type (field-pvar 45) bit-list))
(declare (type (field-pvar 4) found-index))
(declare (type boolean-pvar received-something))
(declare (type boolean-pvar my-address-taken))
(declare (type boolean-pvar dest-fixed))
(declare (type (field-pvar cm:*cube-address-length*) return-address))
(do ((its 1 (1+ its)))
        ((<= (*when (not!! dest-fixed) (*sum (!! (the fixnum 1)))) 200))
      (*when (not!! dest-fixed)
        (format t "~D: computing addresses for ~D processors~%" its
                      (*sum (!! (the fixnum 1))))
        (*set dest-address (!! (the fixnum 0)))
        (*set bit-list (!! (the fixnum 0)))
        (*set found-index (!! (the fixnum 0)))
        (do ()
          ((= 0 (*when (<!! found-index (!! (the fixnum 3)))
                      (*sum (!! (the fixnum 1))))))
          (*when (<!! found-index (!! (the fixnum 3)))
            (*let ((ran-position (random!! (!! (the fixnum 15))))
           (not-unique nil!!))
      (declare (type (field-pvar 5) ran-position))
      (declare (type boolean-pvar not-unique))
      (do ((i 0 (1+ i)))
          ((= i 3))
        (*when (<!! (!! (the fixnum i)) found-index)
          (*when (=!! (load-byte!! bit-list (*!! (!! (the fixnum i))
            (!! (the fixnum 5)))(!! (the fixnum 5)))
      ran-position)
            (*set not-unique t!!))))
      (*when (not!! not-unique)
        (*set bit-list
      (deposit-byte!! bit-list (*!! found-index
                (!! (the fixnum 5))) (!! (the fixnum 5))
      ran-position))
        (*set found-index (1+!! found-index))))))
        (dotimes (i 3)
          (*set dest-address
        (logior!! dest-address
          (ash!! (!! 1)
         (load-byte!! bit-list (*!! (!! (the fixnum i))
                (!! (the fixnum 5)))(!! (the fixnum 5)))))))
        (*let ((bit-sum (!! (the fixnum 0)))
       (dest-address-temp dest-address))
          (declare (type (field-pvar 5) bit-sum))
          (declare (type (field-pvar 15) dest-address-temp))
          (dotimes (i 15)
            (*set bit-sum (+!! bit-sum (load-byte!! dest-address-temp
              (!! (the fixnum 0))(!! (the fixnum 1)))))
            (*set dest-address-temp (ash!! dest-address-temp
                      (!! (the fixnum -1)))))
          (*when (/=!! bit-sum (!! (the fixnum 3)))
            (do-for-selected-processors (cube-add)
      (format t "Processor ~D has a bit-sum not equal to 3~%"
                          cube-add))))
        (*set dest-address (logxor!! dest-address
    (the (field-pvar cm:*cube-address-length*)(self-address!!))))
        (*all (*set received-something nil!!))
        (*pset :overwrite
    (the (field-pvar cm:*cube-address-length*)(self-address!!))
    return-address dest-address received-something)
        (*all
          (*when (and!! received-something (not!! my-address-taken))
            (*set my-address-taken t!!)
            (*pset :no-collisions t!! dest-fixed return-address)))))
(dotimes (i time-loop)
 (format t "~d~%" i)
 (*when dest-fixed
  (format t "~%number active: ~D~%" (*sum (!! (the fixnum 1))))
```

```
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :overwrite v45 v45dest dest-address))
           :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values))))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;; Timing #14
;;; 7/5/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David Myers
;;; Description:  Each PE sends to another PE that is at most a hamming
;;;               distance of 1 away.
;;; Active Processors:    All
;;; Size of Data Used:    45 bit unsigned integers.
;;;
(*defun time-test14 (time-loop test-loop)
 (format *fp* "~%TIME-TEST14~%")
 (format *data* "~%TIME-TEST14~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (mask (!! (the fixnum 0)))
         (dest-address (!! (the fixnum 0))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (declare (type (field-pvar cm:*cube-address-length*) mask))
    (*set mask (logior!! mask (ash!! (!! (the fixnum 1))
          (random!! (!! (the fixnum cm:*cube-address-length*))))))
    (*set dest-address (logxor!! mask (the (field-pvar
          cm:*cube-address-length*)(self-address!!))))
    (dotimes (i time-loop)
      (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
      (cm:time
        (dotimes (j test-loop)
          (*pset :overwrite v45 v45dest dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;; Timing #15
;;; 7/5/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David Myers
;;; Description:  Each PE sends to itself with overwrite.
;;; Active Processors:    All
;;; Size of Data Used:    45 bit unsigned integers.
;;;
(*defun time-test15 (time-loop test-loop)
 (format *fp* "~%TIME-TEST15~%")
 (format *data* "~%TIME-TEST15~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (self (the (field-pvar cm:*cube-address-length*)(self-address!!)))
         (dest-address (!! (the fixnum 0))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (declare (type (field-pvar cm:*cube-address-length*) self))
    (enable-all)
    (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
        (cm:time
```

```
          (dotimes (j test-loop)
            (*pset :overwrite v45 v45dest self))
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
    (concentration self)
    (statistics values time-loop))))
;;;
;;; Timing #16
;;; 7/5/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  Each PE sends to itself with :no-collisions.
;;; Active Processors:      All
;;; Size of Data Used:      45 bit unsigned integers.
;;;
(*defun time-test16 (time-loop test-loop)
 (format *fp* "~%TIME-TEST16~%")
 (format *data* "~%TIME-TEST16~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (self (the (field-pvar cm:*cube-address-length*)(self-address!!)))
         (dest-address (!! (the fixnum 0))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (declare (type (field-pvar cm:*cube-address-length*) self))
    (enable-all)
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
       (cm:time
         (dotimes (j test-loop)
           (*pset :no-collisions v45 v45dest self))
         :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
    (concentration self)
    (statistics values time-loop))))
;;;
;;; Timing #17
;;; 7/5/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  Each PE sends to a random location with overwrite.
;;;               The paris call cm:send-with-overwrite is used.
;;; Active Processors:      All
;;; Size of Data Used:      45 bit unsigned integers.
;;;
(*defun time-test17 (time-loop test-loop)
 (format *fp* "~%TIME-TEST17~%")
 (format *data* "~%TIME-TEST17~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (dest-address (random!! (!! (the fixnum (ash 1
                                   cm:*cube-address-length*))))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (enable-all)
    (let ((v45-addr (pvar-location v45))
          (v45dest-addr (pvar-location v45dest))
          (dest-address-addr (pvar-location dest-address)))
      (declare (type integer v45-addr v45dest-addr dest-address-addr))
      (dotimes (i time-loop)
       (format t "~d~%" i)
       (multiple-value-bind (a cm-time b c)
        (cm:time
         (with-paris-from-*lisp
         (dotimes (j test-loop)
          (cm:send-with-overwrite v45dest-addr dest-address-addr
```

```
                                        v45-addr 45)))
          :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #18
;;; 7/5/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  Each PE sends to (max-PE-# - self-address).  The paris call
;;;               cm:send is used.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers.
;;;
(*defun time-test18 (time-loop test-loop)
  (format *fp* "~%TIME-TEST18~%")
  (format *data* "~%TIME-TEST18~%")
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
          (v45dest (random!! (!! (the fixnum (ash 1 45)))))
          (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
                (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
     (declare (type (field-pvar 45) v45))
     (declare (type (field-pvar 45) v45dest))
     (declare (type (field-pvar cm:*cube-address-length*) dest-address))
     (enable-all)
     (let ((v45-addr (pvar-location v45))
           (v45dest-addr (pvar-location v45dest))
           (dest-address-addr (pvar-location dest-address)))
       (declare (type integer v45-addr v45dest-addr dest-address-addr))
       (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
         (cm:time
          (with-paris-from-*lisp
          (dotimes (j test-loop)
            (cm:send v45dest-addr dest-address-addr v45-addr 45)))
         :return-statistics-only-p t)
         (vector-push (/ cm-time test-loop) values)))
       (concentration dest-address)
       (statistics values time-loop)))))
;;;
;;; Timing #19
;;; 7/5/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  Each PE sends to (- max-pe-address self-address) with
;;;               :no-collisions.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers.
;;;
(*defun time-test19 (time-loop test-loop)
  (format *fp* "~%TIME-TEST19~%")
  (format *data* "~%TIME-TEST19~%")
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
   (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
          (v45dest (random!! (!! (the fixnum (ash 1 45)))))
          (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
                (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
     (declare (type (field-pvar 45) v45))
     (declare (type (field-pvar 45) v45dest))
     (declare (type (field-pvar cm:*cube-address-length*) dest-address))
     (enable-all)
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
          (*pset :no-collisions v45 v45dest dest-address))
       :return-statistics-only-p t)
```

```
      (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;; Timing #20
;;; 7/5/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  Each PE sends to (- max-pe-address self-address) with
;;;               :overwrite
;;; Active Processors:      All
;;; Size of Data Used:      45 bit unsigned integers.
;;;
(*defun time-test20 (time-loop test-loop)
 (format *fp* "~%TIME-TEST20~%")
 (format *data* "~%TIME-TEST20~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
       (v45dest (random!! (!! (the fixnum (ash 1 45)))))
       (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
           (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (enable-all)
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :overwrite v45 v45dest dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;; Timing #21
;;; 7/5/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  All PEs send to a random address, with :overwrite,
;;;               then the receiving PEs (64% of all PES) send data
;;;               back to one of their original sending PEs.
;;; Active Processors:      All
;;; Size of Data Used:      45 bit unsigned integers.
;;;
(*defun time-test21 (time-loop test-loop)
 (format *fp* "~%TIME-TEST21~%")
 (format *data* "~%TIME-TEST21~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
       (v45dest (random!! (!! (the fixnum (ash 1 45)))))
       (rcvd-address (!! (the fixnum 0)))
       (dest-address (random!! (!! (the fixnum (ash 1
                    cm:*cube-address-length*))))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (declare (type (field-pvar cm:*cube-address-length*) rcvd-address))
    (dotimes (i time-loop)
     (enable-all)
     (format t "~d~%" i)
     (*pset :overwrite (the (field-pvar cm:*cube-address-length*)
              (self-address!!)) rcvd-address dest-address)
     (*when (/=!! rcvd-address (!! (the fixnum 0)))
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (dotimes (j test-loop)
         (*pset :no-collisions v45 v45dest rcvd-address))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values))))
```

```
      (concentration dest-address)
      (statistics values time-loop))))
;;;
;;; Timing #22
;;; 7/5/88
;;; Written by:             Bill O'Farrell
;;; Modified by:            David Myers
;;; Description:  Every processor receiving data receives exactly 2
;;;           messages.
;;; Active Processors:         All
;;; Size of Data Used:         45 bit unsigned integers.
;;;
(*defun time-test22 (time-loop test-loop)
 (format *fp* "~%TIME-TEST22~%")
 (format *data* "~%TIME-TEST22~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
        (v45dest (random!! (!! (the fixnum (ash 1 45)))))
        (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
             (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (enable-all)
    (*when (oddp!! dest-address)
     (*set dest-address (1-!! dest-address)))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :overwrite v45 v45dest dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;; Timing #23
;;; 7/5/88
;;; Written by:             Bill O'Farrell
;;; Modified by:            David Myers
;;; Description:  Every processor receiving data receives exactly 4
;;;           messages.
;;; Active Processors:         All
;;; Size of Data Used:         45 bit unsigned integers.
;;;
(*defun time-test23 (time-loop test-loop)
 (format *fp* "~%TIME-TEST23~%")
 (format *data* "~%TIME-TEST23~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
        (v45dest (random!! (!! (the fixnum (ash 1 45)))))
        (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
             (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (enable-all)
    (*set dest-address (the (field-pvar cm:*cube-address-length*)
          (deposit-byte!! dest-address (!! (the fixnum 0))
          (!! (the fixnum 2))(!! (the fixnum 0)))))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :overwrite v45 v45dest dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop))))
```

```
;;;
;;; Timing #24
;;; 7/5/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David Myers
;;; Description:  Every processor receiving data receives exactly 8
;;;               messages.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers.
;;;
(*defun time-test24 (time-loop test-loop)
 (format *fp* "~%TIME-TEST24~%")
 (format *data* "~%TIME-TEST24~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
        (v45dest (random!! (!! (the fixnum (ash 1 45)))))
        (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
              (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (enable-all)
   (*set dest-address (the (field-pvar cm:*cube-address-length*)(deposit-byte!!
       dest-address (!! (the fixnum 0))(!! (the fixnum 3))
       (!! (the fixnum 0)))))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (*pset :overwrite v45 v45dest dest-address))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop))))
;;;
;;; Timing #25
;;; 7/5/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David Myers
;;; Description:  Every processor receiving data receives exactly 16
;;;               messages.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers.
;;;
(*defun time-test25 (time-loop test-loop)
 (format *fp* "~%TIME-TEST25~%")
 (format *data* "~%TIME-TEST25~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
        (v45dest (random!! (!! (the fixnum (ash 1 45)))))
        (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
              (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (enable-all)
   (*set dest-address (the (field-pvar cm:*cube-address-length*)(deposit-byte!!
       dest-address (!! (the fixnum 0))(!! (the fixnum 4))
       (!! (the fixnum 0)))))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (*pset :overwrite v45 v45dest dest-address))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop))))
;;;
```

```
;;; Timing #26
;;; 7/5/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David Myers
;;; Description:  Every processor receiving data receives exactly 32
;;;              messages.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers.
;;;
(*defun time-test26 (time-loop test-loop)
 (format *fp* "~%TIME-TEST26~%")
 (format *data* "~%TIME-TEST26~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
        (v45dest (random!! (!! (the fixnum (ash 1 45)))))
        (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
            (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (enable-all)
   (*set dest-address (the (field-pvar cm:*cube-address-length*)(deposit-byte!!
       dest-address (!! (the fixnum 0))(!! (the fixnum 5))
       (!! (the fixnum 0)))))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
        (*pset :overwrite v45 v45dest dest-address))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop))))
;;;
;;; Timing #27
;;; 7/5/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David Myers
;;; Description:  Every processor receiving data receives exactly 64
;;;              messages.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers.
;;;
(*defun time-test27 (time-loop test-loop)
 (format *fp* "~%TIME-TEST27~%")
 (format *data* "~%TIME-TEST27~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
        (v45dest (random!! (!! (the fixnum (ash 1 45)))))
        (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
            (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (enable-all)
   (*set dest-address (the (field-pvar cm:*cube-address-length*)(deposit-byte!!
       dest-address (!! (the fixnum 0))(!! (the fixnum 6))
       (!! (the fixnum 0)))))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
        (*pset :overwrite v45 v45dest dest-address))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop))))
;;;
;;; Timing #28
```

```
;;; 7/5/88
;;; Written by:              Bill O'Farrell
;;; Modified by:             David Myers
;;; Description:  Every processor receiving data receives exactly 128
;;;               messages.
;;; Active Processors:        All
;;; Size of Data Used:        45 bit unsigned integers.
;;;
(*defun time-test28 (time-loop test-loop)
 (format *fp* "~%TIME-TEST28~%")
 (format *data* "~%TIME-TEST28~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
        (v45dest (random!! (!! (the fixnum (ash 1 45)))))
        (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
             (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (enable-all)
   (*set dest-address (the (field-pvar cm:*cube-address-length*)(deposit-byte!!
        dest-address (!! (the fixnum 0))(!! (the fixnum 7))
        (!! (the fixnum 0)))))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (*pset :overwrite v45 v45dest dest-address))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop))))
;;;
;;; Timing #29
;;; 7/5/88
;;; Written by:              Bill O'Farrell
;;; Modified by:             David Myers
;;; Description:  Every processor receiving data receives exactly 256
;;;               messages.
;;; Active Processors:        All
;;; Size of Data Used:        45 bit unsigned integers.
;;;
(*defun time-test29 (time-loop test-loop)
 (format *fp* "~%TIME-TEST29~%")
 (format *data* "~%TIME-TEST29~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
        (v45dest (random!! (!! (the fixnum (ash 1 45)))))
        (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
             (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (enable-all)
   (*set dest-address (the (field-pvar cm:*cube-address-length*)(deposit-byte!!
        dest-address (!! (the fixnum 0))(!! (the fixnum 8))
        (!! (the fixnum 0)))))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (*pset :overwrite v45 v45dest dest-address))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop))))
;;;
;;; Timing 30
;;; 7/5/88
```

```
;;; Written by:            Bill O'Farrell
;;; Modified by:           David W. Myers
;;; Description:  This program times a send with overwrite of two
;;;               10 bit unsigned integers to random locations
;;; Active Processors:      All
;;; Size of Data Used:      10 bit unsigned integers
;;;
(*defun time-test30 (time-loop test-loop)
 (format *fp* "~%TIME-TEST30~%")
 (format *data* "~%TIME-TEST30~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v10 (random!! (!! (the fixnum (ash 1 10)))))
         (v10dest (random!! (!! (the fixnum (ash 1 10)))))
         (dest-address (random!! (!! (the fixnum (ash 1
                                  cm:*cube-address-length*))))))
   (declare (type (field-pvar 10) v10))
   (declare (type (field-pvar 10) v10dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (enable-all)
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (*pset :overwrite v10 v10dest dest-address))
           :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop))))
;;;
;;; Timing #31
;;; 7/5/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  Each PE sends to (- max-pe-address self-address) with
;;;               :no-collisions specified within *pset.
;;; Active Processors:      All
;;; Size of Data Used:      10 bit unsigned integers.
;;;
(*defun time-test31 (time-loop test-loop)
 (format *fp* "~%TIME-TEST31~%")
 (format *data* "~%TIME-TEST31~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v10 (random!! (!! (the fixnum (ash 1 10)))))
         (v10dest (random!! (!! (the fixnum (ash 1 10)))))
         (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
            (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
   (declare (type (field-pvar 10) v10))
   (declare (type (field-pvar 10) v10dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (enable-all)
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (*pset :no-collisions v10 v10dest dest-address))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop))))
;;;
;;; Timing #32
;;; 7/5/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  2048 processors all send to random locations
;;;               and :overwrite is specified. This program is written
;;;               for 8k processors. If more than 8k processors are used,
;;;               then the (*when ...) expression must be changed.
;;; Active Processors:      self-address mod 4 = 0
```

```
;;; Size of Data Used:        45 bit unsigned integers.
;;;
(*defun time-test32 (time-loop test-loop)
 (format *fp* "~%TIME-TEST32~%")
 (format *data* "~%TIME-TEST32~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (dest-address (random!! (!! (the fixnum
                         (ash 1 cm:*cube-address-length*))))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (dotimes (i time-loop)
    (enable-all)
    (format t "~d~%" i)
    (*when (=!! (!! (the fixnum 0))(mod!! (the (field-pvar
           cm:*cube-address-length*)(self-address!!))(!! (the fixnum 4))))
     (format t "~%number of processors: ~D~%"(cm:global-count cm:context-flag))
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :overwrite v45 v45dest dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values))))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;; Timing #33
;;; 7/6/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David Myers
;;; Description:  2048 processors send to locations (max-PE-# - self-address)
;;;              and :no-collisions is specified.  This program is written
;;;              for 8k processors.  If more than 8k processors are used,
;;;              then the (*when ...) expression must be changed.
;;; Active Processors:       self-address mod 4 = 0
;;; Size of Data Used:       45 bit unsigned integers.
;;;
(*defun time-test33 (time-loop test-loop)
 (format *fp* "~%TIME-TEST33~%")
 (format *data* "~%TIME-TEST33~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
             (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (dotimes (i time-loop)
    (enable-all)
    (format t "~d~%" i)
    (*when (=!! (!! (the fixnum 0))(mod!! (the (field-pvar
           cm:*cube-address-length*)(self-address!!))(!! (the fixnum 4))))
     (format t "~%number of processors: ~D~%"(cm:global-count cm:context-flag))
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :no-collisions v45 v45dest dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values))))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;; Timing #34
;;; 7/6/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David Myers
;;; Description:  2048 processors send to locations (max-PE-# - self-address)
;;;              and :overwrite is specified.  This program is written
```

```
;;;             for 8k processors.  If more than 8k processors are used,
;;;             then the (*when ...) expression must be changed.
;;; Active Processors:          self-address mod 4 = 0
;;; Size of Data Used:          45 bit unsigned integers.
;;;
(*defun time-test34 (time-loop test-loop)
 (format *fp* "~%TIME-TEST34~%")
 (format *data* "~%TIME-TEST34~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
              (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (enable-all)
   (*when (=!! (!! (the fixnum 0))(mod!! (the (field-pvar
              cm:*cube-address-length*)(self-address!!))(!! (the fixnum 4))))
    (dotimes (i time-loop)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :overwrite v45 v45dest dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
     (concentration dest-address))
    (statistics values time-loop))))
;;; Timing #35
;;; 7/6/88
;;; Written by:             Bill O'Farrell
;;; Modified by:            David Myers
;;; Description:   2048 processors all send to random locations
;;;             and :overwrite is specified.  This program is written
;;;             for 8k processors.  If more than 8k processors are used,
;;;             then the (*when ...) expression must be changed.
;;; Active Processors:          self-address mod 4 = 0
;;; Size of Data Used:          10 bit unsigned integers.
;;;
(*defun time-test35 (time-loop test-loop)
 (format *fp* "~%TIME-TEST35~%")
 (format *data* "~%TIME-TEST35~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v10 (random!! (!! (the fixnum (ash 1 10)))))
         (v10dest (random!! (!! (the fixnum (ash 1 10)))))
         (dest-address (random!! (!! (the fixnum
                      (ash 1 cm:*cube-address-length*))))))
   (declare (type (field-pvar 10) v10))
   (declare (type (field-pvar 10) v10dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (dotimes (i time-loop)
    (enable-all)
    (format t "~d~%" i)
    (*when (=!! (!! (the fixnum 0))(mod!! (the (field-pvar
             cm:*cube-address-length*)(self-address!!))(!! (the fixnum 4))))
     (format t "~%number of processors: ~D~%"(cm:global-count cm:context-flag))
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*pset :overwrite v10 v10dest dest-address))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values))))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;; Timing #39
;;; 7/6/88
;;; Written by:             Bill O'Farrell
;;; Modified by:            David Myers
;;; Description:   512 processors all send to random locations
;;;             and :overwrite is specified.  This program is written
```

```
;;;             for 8k processors.  If more than 8k processors are used,
;;;             then the (*when ...) expression must be changed.
;;; Active Processors:          self-address mod 4 = 0
;;; Size of Data Used:          10 bit unsigned integers.
;;;
(*defun time-test39 (time-loop test-loop)
 (format *fp* "~%TIME-TEST39~%")
 (format *data* "~%TIME-TEST39~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v10 (random!! (!! (the fixnum (ash 1 10)))))
         (v10dest (random!! (!! (the fixnum (ash 1 10)))))
         (dest-address (random!! (!! (the fixnum
                           (ash 1 cm:*cube-address-length*))))))
    (declare (type (field-pvar 10) v10))
    (declare (type (field-pvar 10) v10dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (dotimes (i time-loop)
      (enable-all)
      (format t "~d~%" i)
      (*when (=!! (!! (the fixnum 0))(mod!! (the (field-pvar
              cm:*cube-address-length*)(self-address!!))(!! (the fixnum 16))))
        (format t "~%number of processors: ~D~%"(cm:global-count cm:context-flag))
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*pset :overwrite v10 v10dest dest-address))
            :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values))))
      (concentration dest-address)
      (statistics values time-loop))))
;;;
;;; Timing #45
;;; 7/5/88
;;; Written by:               Bill O'Farrell
;;; Modified by:              David Myers
;;; Description:  Each PE reads from a random location.
;;;             The paris call cm:get is used.
;;; Active Processors:          All
;;; Size of Data Used:          45 bit unsigned integers.
;;;
(*defun time-test45 (time-loop test-loop)
 (format *fp* "~%TIME-TEST45~%")
 (format *data* "~%TIME-TEST45~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (dest-address (random!! (!! (the fixnum (ash 1
                                   cm:*cube-address-length*))))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (let ((v45-addr (pvar-location v45))
          (v45dest-addr (pvar-location v45dest))
          (dest-address-addr (pvar-location dest-address)))
      (declare (type integer v45-addr v45dest-addr dest-address-addr))
      (dotimes (i time-loop)
        (enable-all)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (with-paris-from-*lisp
            (dotimes (j test-loop)
              (cm:get v45dest-addr dest-address-addr v45-addr 45)))
           :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #47
;;; 7/5/88
;;; Written by:               Bill O'Farrell
```

```
;;; Modified by:           David Myers
;;; Description:  Each PE reads from a random location.
;;;               A *lisp call is used.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers.
;;;
(*defun time-test47 (time-loop test-loop)
 (format *fp* "~%TIME-TEST47~%")
 (format *data* "~%TIME-TEST47~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (!! (the fixnum 0)))
         (dest-address (random!! (!! (the fixnum (ash 1
                               cm:*cube-address-length*))))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (enable-all)
    (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
        (cm:time
          (dotimes (j test-loop)
            (*set v45dest (pref!! v45 dest-address :collisions-allowed)))
         :return-statistics-only-p t)
         (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop))))
;;;
;;; Timing #48
;;; 7/5/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  Each PE reads from a random location.
;;;               A *lisp call is used.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers.
;;;
(*defun time-test48 (time-loop test-loop)
 (format *fp* "~%TIME-TEST48~%")
 (format *data* "~%TIME-TEST48~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (!! (the fixnum 0)))
         (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
              (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (enable-all)
    (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
        (cm:time
          (dotimes (j test-loop)
            (*set v45dest (pref!! v45 dest-address :no-collisions)))
         :return-statistics-only-p t)
         (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop)))))
;;;
;;; Timing #49
;;; 7/6/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  Each PE reads from a unique location.
;;;               cm:get is used.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers.
;;;
(*defun time-test49 (time-loop test-loop)
```

C - 2

```
 (format *fp* "~%TIME-TEST49~%")
 (format *data* "~%TIME-TEST49~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
              (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (let ((v45-addr (pvar-location v45))
         (v45dest-addr (pvar-location v45dest))
         (dest-address-addr (pvar-location dest-address)))
    (declare (type integer v45-addr v45dest-addr dest-address-addr))
    (dotimes (i time-loop)
     (enable-all)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (with-paris-from-*lisp
        (dotimes (j test-loop)
        (cm:get v45dest-addr dest-address-addr v45-addr 45)))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop)))))
;;;
;;; Timing #50
;;; 7/6/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David Myers
;;; Description:  Every processor receiving a read request, receives
;;;               exactly 2 request.  cm:get is used.
;;; Active Processors:      All
;;; Size of Data Used:      45 bit unsigned integers.
;;;
(*defun time-test50 (time-loop test-loop)
 (format *fp* "~%TIME-TEST50~%")
 (format *data* "~%TIME-TEST50~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
              (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (declare (type (field-pvar cm:*cube-address-length*) dest-address))
   (enable-all)
   (*when (oddp!! dest-address)
    (*set dest-address (1-!! dest-address)))
   (let ((v45-addr (pvar-location v45))
         (v45dest-addr (pvar-location v45dest))
         (dest-address-addr (pvar-location dest-address)))
    (declare (type integer v45-addr v45dest-addr dest-address-addr))
    (dotimes (i time-loop)
     (enable-all)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (with-paris-from-*lisp
        (dotimes (j test-loop)
        (cm:get v45dest-addr dest-address-addr v45-addr 45)))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (concentration dest-address)
   (statistics values time-loop)))))
;;;
;;; Timing #51
;;; 7/6/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David Myers
```

```
;;; Description:  Every processor receiving a read request, receives
;;;               exactly 4 request.  cm:get is used.
;;; Active Processors:        All
;;; Size of Data Used:        45 bit unsigned integers.
;;;
(*defun time-test51 (time-loop test-loop)
 (format *fp* "~%TIME-TEST51~%")
 (format *data* "~%TIME-TEST51~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
              (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (enable-all)
    (*set dest-address (the (field-pvar cm:*cube-address-length*)
             (deposit-byte!! dest-address (!! (the fixnum 0))
             (!! (the fixnum 2))(!! (the fixnum 0)))))
    (let ((v45-addr (pvar-location v45))
          (v45dest-addr (pvar-location v45dest))
          (dest-address-addr (pvar-location dest-address)))
      (declare (type integer v45-addr v45dest-addr dest-address-addr))
      (dotimes (i time-loop)
        (enable-all)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (with-paris-from-*lisp
              (dotimes (j test-loop)
              (cm:get v45dest-addr dest-address-addr v45-addr 45)))
          :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values)))
    (concentration dest-address)
    (statistics values time-loop)))))
;;;
;;; Timing #52
;;; 7/6/88
;;; Written by:              Bill O'Farrell
;;; Modified by:             David Myers
;;; Description:  Every processor receiving a read request, receives
;;;               exactly 16 request.  cm:get is used.
;;; Active Processors:        All
;;; Size of Data Used:        45 bit unsigned integers.
;;;
(*defun time-test52 (time-loop test-loop)
 (format *fp* "~%TIME-TEST52~%")
 (format *data* "~%TIME-TEST52~%")
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45)))))
         (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
              (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (declare (type (field-pvar cm:*cube-address-length*) dest-address))
    (enable-all)
    (*set dest-address (the (field-pvar cm:*cube-address-length*)
             (deposit-byte!! dest-address (!! (the fixnum 0))
             (!! (the fixnum 4))(!! (the fixnum 0)))))
    (let ((v45-addr (pvar-location v45))
          (v45dest-addr (pvar-location v45dest))
          (dest-address-addr (pvar-location dest-address)))
      (declare (type integer v45-addr v45dest-addr dest-address-addr))
      (dotimes (i time-loop)
        (enable-all)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (with-paris-from-*lisp
```

```
            (dotimes (j test-loop)
            (cm:get v45dest-addr dest-address-addr v45-addr 45)))
          :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #53
;;; 7/6/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David Myers
;;; Description:  Every processor receiving a read request, receives
;;;               exactly 256 request.  cm:get is used.
;;; Active Processors:       All
;;; Size of Data Used:       45 bit unsigned integers.
;;;
(*defun time-test53 (time-loop test-loop)
  (format *fp* "~%TIME-TEST53~%")
  (format *data* "~%TIME-TEST53~%")
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
    (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
           (v45dest (random!! (!! (the fixnum (ash 1 45)))))
           (dest-address (-!! (!! (the fixnum (- cm:*user-cube-address-limit* 1)))
                (the (field-pvar cm:*cube-address-length*)(self-address!!)))))
      (declare (type (field-pvar 45) v45))
      (declare (type (field-pvar 45) v45dest))
      (declare (type (field-pvar cm:*cube-address-length*) dest-address))
      (enable-all)
      (*set dest-address (the (field-pvar cm:*cube-address-length*)
                (deposit-byte!! dest-address (!! (the fixnum 0))
                (!! (the fixnum 8))(!! (the fixnum 0)))))
      (let ((v45-addr (pvar-location v45))
            (v45dest-addr (pvar-location v45dest))
            (dest-address-addr (pvar-location dest-address)))
        (declare (type integer v45-addr v45dest-addr dest-address-addr))
        (dotimes (i time-loop)
          (enable-all)
          (format t "~d~%" i)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (with-paris-from-*lisp
                (dotimes (j test-loop)
                (cm:get v45dest-addr dest-address-addr v45-addr 45)))
          :return-statistics-only-p t)
          (vector-push (/ cm-time test-loop) values)))
      (concentration dest-address)
      (statistics values time-loop)))))
;;;
;;; Timing #54
;;; 7/6/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David W. Myers
;;; Description:  Timing of an additive scan with 16 bit numbers.
;;; Active Processors:       All
;;; Size of Data Used:       16 bit unsigned integers
;;;
(*defun time-test54 (time-loop test-loop)
  (format *fp* "~%TIME-TEST54~%")
  (format *data* "~%TIME-TEST54~%")
  (enable-all)
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
    (*let ((fvar (random!! (!! (the fixnum (ash 1 5)))))
           (rvar (!! (the fixnum 0))))
      (declare (type (field-pvar 16) fvar))
      (declare (type (field-pvar 16) rvar))
      (dotimes (i time-loop)
        (format t "~d~%" i)
        (multiple-value-bind (a cm-time b c)
          (cm:time
            (dotimes (j test-loop)
              (*set rvar (scan!! fvar '+!!)))
```

```
          :return-statistics-only-p t)
        (vector-push (/ cm-time test-loop) values)))
      (statistics values time-loop))))
;;;
;;; Timing #55
;;; 7/6/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David W. Myers
;;; Description:  Timing of a max scan with 16 bit numbers.
;;; Active Processors:     All
;;; Size of Data Used:     16 bit unsigned integers
;;;
(*defun time-test55 (time-loop test-loop)
 (format *fp* "~%TIME-TEST55~%")
 (format *data* "~%TIME-TEST55~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((fvar (random!! (!! (the fixnum (ash 1 16)))))
         (rvar (!! (the fixnum 0))))
    (declare (type (field-pvar 16) fvar))
    (declare (type (field-pvar 16) rvar))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*set rvar (scan!! fvar 'max!!)))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (statistics values time-loop))))
;;;
;;; Timing #56
;;; 7/6/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David W. Myers
;;; Description:  Timing of an additive scan with 16 bit numbers.
;;;               The paris call cm:unsigned-plus-scan is used.
;;; Active Processors:     All
;;; Size of Data Used:     16 bit unsigned integers
;;;
(*defun time-test56 (time-loop test-loop)
 (format *fp* "~%TIME-TEST56~%")
 (format *data* "~%TIME-TEST56~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((src-var (random!! (!! (the fixnum (ash 1 5)))))
         (dest-var (!! (the fixnum 0))))
    (declare (type (field-pvar 16) src-var))
    (declare (type (field-pvar 16) dest-var))
    (let ((src-var-addr (pvar-location src-var))
          (dest-var-addr (pvar-location dest-var)))
     (declare (type integer src-var-addr dest-var-addr))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (with-paris-from-*lisp
        (dotimes (j test-loop)
         (cm:unsigned-plus-scan dest-var-addr src-var-addr 23 8)))
        :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
     (statistics values time-loop)))))
;;;
;;; Timing #57
;;; 7/6/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David W. Myers
;;; Description:  Timing of a max scan with 16 bit numbers.
;;;               The paris call cm:unsigned-max-scan is used.
;;; Active Processors:     All
;;; Size of Data Used:     16 bit unsigned integers
```

```
;;;
(*defun time-test57 (time-loop test-loop)
 (format *fp* "~%TIME-TEST57~%")
 (format *data* "~%TIME-TEST57~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((src-var (random!! (!! (the fixnum (ash 1 5)))))
         (dest-var (!! (the fixnum 0))))
   (declare (type (field-pvar 16) src-var))
   (declare (type (field-pvar 16) dest-var))
   (let ((src-var-addr (pvar-location src-var))
         (dest-var-addr (pvar-location dest-var)))
    (declare (type integer src-var-addr dest-var-addr))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (with-paris-from-*lisp
       (dotimes (j test-loop)
        (cm:unsigned-max-scan dest-var-addr src-var-addr 16)))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (statistics values time-loop)))))
;;;
;;; Timing #58
;;; 7/6/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David Myers
;;; Description:  This program times the ranking operation.
;;;               The *lisp function rank!! is used.
;;; Active Processors:
;;; Size of Data Used:
;;;
(*defun time-test58 (time-loop test-loop)
 (format *fp* "~%TIME-TEST58~%")
 (format *data* "~%TIME-TEST58~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((fvar (random!! (!! (the fixnum (ash 1 16))))))
   (declare (type (field-pvar 16) fvar))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (rank!! fvar '<=!!))
      :return-statistics-only-p t)
     (vector-push (/ cm-time test-loop) values)))
   (statistics values time-loop))))
;;;
;;; Timing #59
;;; 7/6/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David Myers
;;; Description:  Time the counting of the css.
;;; Active Processors:
;;; Size of Data Used:
;;;
(*defun time-test59 (time-loop test-loop)
 (format *fp* "~%TIME-TEST59~%")
 (format *data* "~%TIME-TEST59~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (dotimes (i time-loop)
   (format t "~d~%" i)
   (multiple-value-bind (a cm-time b c)
    (cm:time
     (dotimes (j test-loop)
      (cm:global-count cm:context-flag))
     :return-statistics-only-p t)
```

```
    (vector-push (/ cm-time test-loop) values)))
  (statistics values time-loop)))
;;; Timing #60
;;; 7/6/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  Every processor fetches a 45 bit unsigned integer
;;;               from a random processor.  Each fetch to a random PE
;;;               can access one of  a possible 64 locations in memory.
;;; Active Processors:      All
;;; Size of Data Used:      45 bit unsigned integers.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Timing #63
;;; 7/6/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David Myers
;;; Description:  Each PE sends shifts a 45 bit unsigned integer a random
;;;               number of positions.  The paris call cm:shift is used.
;;; Active Processors:      All
;;; Size of Data Used:      45 bit unsigned integers.
;;;
(*defun time-test63 (time-loop test-loop)
  (format *fp* "~%TIME-TEST63~%")
  (format *data* "~%TIME-TEST63~%")
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
    (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
           (v45shift (random!! (!! (the fixnum 45)))))
      (declare (type (field-pvar 45) v45))
      (declare (type (field-pvar 6) v45shift))
      (enable-all)
      (let ((v45-addr (pvar-location v45))
            (v45shift-addr (pvar-location v45shift)))
        (declare (type integer v45-addr v45shift-addr))
        (dotimes (i time-loop)
          (format t "~d~%" i)
          (multiple-value-bind (a cm-time b c)
            (cm:time
              (with-paris-from-*lisp
              (dotimes (j test-loop)
                (cm:shift v45-addr v45shift-addr 45 6)))
             :return-statistics-only-p t)
            (vector-push (/ cm-time test-loop) values)))
        (statistics values time-loop)))))
;;;
;;; Timing #65
;;; 7/6/88
;;; Written by:            Bill O'Farrell
;;; Modified by:           David W. Myers
;;; Description:  A load-byte of 34 bits taken from a 748 bit field is tested.
;;;               The paris call cm:aref is used.
;;; Active Processors:      All
;;; Size of Data Used:      748 bit unsigned integers.
;;;
(*defun time-test65 (time-loop test-loop)
  (format *fp* "~%TIME-TEST65~%")
  (format *data* "~%TIME-TEST65~%")
  (enable-all)
  (setq values (make-array '(100) :fill-pointer 0))
  (*all
    (*let ((v748 (random!! (!! (the fixnum (ash 1 748)))))
           (vdest (!! (the fixnum 0)))
           (index (!! (the fixnum 22))))
      (declare (type (field-pvar 748) v748))
      (declare (type (field-pvar 34) vdest))
      (declare (type (field-pvar 5) index))
      (enable-all)
      (let ((v748-addr (pvar-location v748))
            (vdest-addr (pvar-location vdest))
            (index-addr (pvar-location index)))
        (declare (type integer v748-addr vdest-addr index-addr))
```

```
    (dotimes (i time-loop)
     (enable-all)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (cm:aref vdest-addr v748-addr index-addr 34 5 22 34))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (statistics values time-loop)))))
;;;
;;; Timing #67
;;; 7/6/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David W. Myers
;;; Description:
;;;
;;; Active Processors:     All
;;; Size of Data Used:     1240 bit unsigned integers.
;;;
(*defun time-test67 (time-loop test-loop)
 (format *fp* "~%TIME-TEST67~%")
 (format *data* "~%TIME-TEST67~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v1240 (random!! (!! (the fixnum (ash 1 748)))))
        (vdest (!! (the fixnum 0)))
        (index (!! (the fixnum 22))))
   (declare (type (field-pvar 748) v1240))
   (declare (type (field-pvar 34) vdest))
   (declare (type (field-pvar 5) index))
   (enable-all)
   (let ((v1240-addr (pvar-location v1240))
         (vdest-addr (pvar-location vdest))
         (index-addr (pvar-location index)))
    (declare (type integer v1240-addr vdest-addr index-addr))
    (dotimes (i time-loop)
     (enable-all)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (cm:aref vdest-addr v1240-addr index-addr 31 6 40 31))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (statistics values time-loop)))))
;;;
;;; Timing #71
;;; 7/6/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David W. Myers
;;; Description:  This program times the movement of one pvar to another pvar.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers.
;;;
(*defun time-test71 (time-loop test-loop)
 (format *fp* "~%TIME-TEST71~%")
 (format *data* "~%TIME-TEST71~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
        (v45dest (random!! (!! (the fixnum (ash 1 45))))))
   (declare (type (field-pvar 45) v45))
   (declare (type (field-pvar 45) v45dest))
   (dotimes (i time-loop)
    (format t "~d~%" i)
    (multiple-value-bind (a cm-time b c)
     (cm:time
      (dotimes (j test-loop)
       (*set v45dest v45))
      :return-statistics-only-p t)
```

```
      (vector-push (/ cm-time test-loop) values)))
    (statistics values time-loop))))
;;;
;;; Timing #72
;;; 7/6/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David W. Myers
;;; Description:  This program times the movement of one pvar to another pvar.
;;;               The paris function cm:move is used.
;;; Active Processors:      All
;;; Size of Data Used:      45 bit unsigned integers.
;;;
(*defun time-test72 (time-loop test-loop)
 (format *fp* "~%TIME-TEST72~%")
 (format *data* "~%TIME-TEST72~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v45 (random!! (!! (the fixnum (ash 1 45)))))
         (v45dest (random!! (!! (the fixnum (ash 1 45))))))
    (declare (type (field-pvar 45) v45))
    (declare (type (field-pvar 45) v45dest))
    (let ((v45-addr (pvar-location v45))
          (v45dest-addr (pvar-location v45dest)))
     (declare (type integer v45-addr v45dest-addr))
     (dotimes (i time-loop)
      (format t "~d~%" i)
      (multiple-value-bind (a cm-time b c)
       (cm:time
        (with-paris-from-*lisp
         (dotimes (j test-loop)
          (cm:move v45dest-addr v45-addr 45)))
       :return-statistics-only-p t)
       (vector-push (/ cm-time test-loop) values)))
     (statistics values time-loop)))))
;;;
;;; Timing #73
;;; 7/6/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David W. Myers
;;; Description:  This program times the movement of one pvar to another pvar.
;;; Active Processors:      All
;;; Size of Data Used:      150 bit unsigned integers.
;;;
(*defun time-test73 (time-loop test-loop)
 (format *fp* "~%TIME-TEST73~%")
 (format *data* "~%TIME-TEST73~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((v150 (random!! (!! (the fixnum (ash 1 150)))))
         (v150dest (random!! (!! (the fixnum (ash 1 150))))))
    (declare (type (field-pvar 150) v150))
    (declare (type (field-pvar 150) v150dest))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*set v150dest v150))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (statistics values time-loop))))
;;;
;;; Timing #74
;;; 7/6/88
;;; Written by:           Bill O'Farrell
;;; Modified by:          David W. Myers
;;; Description:  This programs the amount of time it take to do a floating
;;;               point addition.
;;; Active Processors:      All
;;; Size of Data Used:      double-float
;;;
```

```
(*defun time-test74 (time-loop test-loop)
 (format *fp* "~%TIME-TEST74~%")
 (format *data* "~%TIME-TEST74~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((x (!! (the double-float 4.483624498)))
         (y (!! (the double-float 3.141592654))))
    (declare (type double-float-pvar x))
    (declare (type double-float-pvar y))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (*set x (!! (the double-float 4.483624498)))
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*set x (+!! x y)))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (statistics values time-loop))))
;;;
;;; Timing #75
;;; 7/6/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David W. Myers
;;; Description:  This programs reports the time required to do a floating
;;;               point multiply.
;;; Active Processors:     All
;;; Size of Data Used:     double-float
;;;
(*defun time-test75 (time-loop test-loop)
 (format *fp* "~%TIME-TEST75~%")
 (format *data* "~%TIME-TEST75~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((x (!! (the double-float 4.483624498)))
         (y (!! (the double-float 3.141592654))))
    (declare (type double-float-pvar x))
    (declare (type double-float-pvar y))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (*set x (!! (the double-float 4.483624498)))
     (multiple-value-bind (a cm-time b c)
      (cm:time
       (dotimes (j test-loop)
        (*set x (*!! x y)))
       :return-statistics-only-p t)
      (vector-push (/ cm-time test-loop) values)))
    (statistics values time-loop))))
;;;
;;; Timing #76
;;; 7/6/88
;;; Written by:          Bill O'Farrell
;;; Modified by:         David W. Myers
;;; Description:  This programs reports the time required to do a global
;;;               sum of unsigned integers.
;;; Active Processors:     All
;;; Size of Data Used:     45 bit unsigned integers
;;;
(*defun time-test76 (time-loop test-loop)
 (format *fp* "~%TIME-TEST76~%")
 (format *data* "~%TIME-TEST76~%")
 (enable-all)
 (setq values (make-array '(100) :fill-pointer 0))
 (*all
  (*let ((fvar (random!! (!! (the fixnum (ash 1 5))))))
   (declare (type (field-pvar 16) fvar))
   (let ((host-var 0))
    (dotimes (i time-loop)
     (format t "~d~%" i)
     (setq host-var 0)
     (multiple-value-bind (a cm-time b c)
```

```
    (cm:time
     (dotimes (j test-loop)
       (setq host-var (*sum fvar)))
     :return-statistics-only-p t)
    (vector-push (/ cm-time test-loop) values)))
(statistics values time-loop)))))
```

Appendix C

Batch Command Utilities

```
#
@ srt_hr = $argv[1]
@ srt_mn = $argv[2]
@ continue = 1
set directory = 'pwd'
@ ck_hour = 0
@ ck_min = 0
set time = 'date | awk ' { print $4 } ''
@ hours = 'echo $time | sed 's/:..:..//''
@ minutes = 'echo $time | sed 's/..://' | sed 's/:..//''
while ( $continue == 1 )
 sleep 900
 @ prev_min = $minutes
 @ prev_hour = $hours
 set time = 'date | awk ' { print $4 } ''
 @ hours = 'echo $time | sed 's/:..:..//''
 @ minutes = 'echo $time | sed 's/..://' | sed 's/:..//''
 @ hr_plus = ( $prev_hour + 1 ) % 24
 @ mn_plus = ( $srt_mn + 15 ) % 60
 if ( ($srt_hr == $hours) && ($srt_mn == $minutes) ) then
  @ continue = 0
 else if (($srt_hr == $hours) && ($srt_mn < ( 15 + $minutes ))) then
  @ continue = 0
 endif
end
date >&! /tmp/trash&
CMrun >&! /tmp/trash&
date >&! /tmp/trash&
```

```
echo "(load \"foo\")" | starlisp
```

```
#
@ srt_hr = $argv[1]
@ srt_mn = $argv[2]
@ continue = 1
set directory = `pwd`
@ ck_hour = 0
@ ck_min = 0
set time = `date | awk ' { print $4 } '`
@ hours = `echo $time | sed 's/:..:..//'`
@ minutes = `echo $time | sed 's/..://' | sed 's/:..//'`
while ( $continue == 1 )
 sleep 900
 @ prev_min = $minutes
 @ prev_hour = $hours
 set time = `date | awk ' { print $4 } '`
 @ hours = `echo $time | sed 's/:..:..//'`
 @ minutes = `echo $time | sed 's/..://' | sed 's/:..//'`
 @ hr_plus = ( $prev_hour + 1 ) % 24
 @ mn_plus = ( $srt_mn + 15 ) % 60
 if ( ($srt_hr == $hours) && ($srt_mn == $minutes) ) then
  @ continue = 0
 else if (($srt_hr == $hours) && ($srt_mn < ( 15 + $minutes ))) then
  @ continue = 0
 endif
end
date >&! /tmp/trash&
old-CMrun >&! /tmp/trash&
date >&! /tmp/trash&
```

```
echo "(load \"foo\")" | old-starlisp
```

```
(in-package '*lisp)
(*cold-boot)
(load "FILE-NAME.vbin")
(main)
(system:quit)
```